

HANDS ON BASIC

Programming

by
Neil Bennett, Ph.D.

Manual by
Scot Kamins, Ph. D.



The Science of Learning

Program copyright © 1983 by Neil Bennett, Ph.D.
Documentation copyright © 1983 by Scot Kamins, Ph.D.

All rights reserved. Any reproduction of the program diskette or this printed documentation is strictly forbidden without the expressed written consent of Edu-Ware Services, Inc.

WARNING: subject to the provisions of the copyright act of 1980, as specified in Public Law 94-553, dated 12 December, 1980 (94 STAT. 3028-29) and amended as Public Law 96-517, the duplication of computer programs without prior consent of the publisher, for the purpose of barter, trade, sale or exchange is a criminal offense, for which the offender may be subject to fine, imprisonment, and/or civil suit. Under the provisions of Section 117 of Public Law 96-517 it is not an infringement for the owner of a computer program to make or authorize the making of another copy or adaptation of that computer program provided that such new copy or adaptation is created for archival purposes only and that all archival copies are destroyed in the event that continued possession of the computer program should cease to be rightful.

This manual was written by Dr. Scot Kamins of Technology Translated, Inc., San Francisco, California.

HANDS ON BASIC was developed by Neil Bennett in cooperation with Edu-Ware Services, Inc. a California software development company dedicated to the production of instructionally valid Computer Aided Instruction and intellectually challenging games.

Edu-Ware Services, Inc.
P.O. Box 22222
Agoura Hills, California 91301-0522

ACKNOWLEDGEMENTS

It is a pleasure to recognize the contributions of those who have helped to bring HANDS ON BASIC to its current state of development. I wish to express my indebtedness to Rudi Hoess of Sydney, Australia for providing much guidance and encouragement; Roger Keating of Sydney, Australia for giving much advice on the educational aspects of the system; Juris Reinfelds and Richard Miller of the University of Wollongong, Australia for contributing many valuable suggestions; and Portia Isaacson, Egil Juliussen and Harold Kinne of Dallas, Texas, and Ted Perry of Sacramento, California for their encouragement.

I would also like to thank the team at Edu-Ware Services, Inc. for helping to make the system into a product.

I am grateful to my wife Judy for her patience and understanding.

N. W. Bennett
February, 1983

INTRODUCTION 7

Section One: COMPUTER THINKING

Chapter 1: Automatic Arithmetic	13
Review	21
Exercises	22
Chapter 2: Computer Variables	23
Review	28
Exercises	28

Section Two: HOBASIC TUTORIAL

Chapter 3: Looping Around In HOB	31
Review	41
Exercises	41
Chapter 4: Limited Looping	43
Review	53
Exercises	53
Chapter 5: The READ/DATA/RESTORE Command Set	54
Review	62
Exercises	62
Chapter 6: Inputs and Interactive Programming	64
Review	72
Exercises	72
Chapter 7: Arrays	74
Review	82
Exercises	83
Chapter 8: Other Ways of Branching	84
Review	94
Exercises	94
Chapter 9: Getting Functional	96
Review	105
Exercises	105

Section Three: PUTTING HOB TO WORK

- 109 Chapter 10: Program Planning**
- 119 Chapter 11: Getting the Bugs Out**

APPENDICES

- 135 A: Immediate Commands**
- 139 B: Deferred Commands**
- 143 C: Built-In Functions**
- 145 D: Errors and Possible Solutions**
- 153 E: Program Tracking Screens**
- 155 F: Summary of System Limits**
- 156 G: Glossary of Terms**
- 161 H: Editing Keys and Commands**
- 162 I: Precedence**
- 163 J: Solutions to Practice Exercises**
- 175 K: Single Key Commands**

- 179 INDEX**

INTRODUCTION

HANDS ON BASIC (HOB), the first in EDU-WARE's Visible Computer Series, is designed to help you, the novice programmer, learn the elements of the computer language BASIC quickly and painlessly. More than just a language, HOB is a language system incorporating tutorial features found nowhere else in the computer literacy world. Its many special commands, tracing screens, error-finding tools and special displays make learning to plan, code and "debug" programs both entertaining and enjoyable.

HOB takes the mystery out of programming by showing you how a computer language processes information. Its unique displays and interactive commands let you watch how the computer "thinks through" your programs every step of the way at a speed and level of complexity that *you* control. Not only does HOB help you learn how to program, it helps you understand the internal workings of BASIC. You end up with a clear grasp of programming from both a theoretical and practical point of view.

The purpose of HANDS ON BASIC is to teach you how to program in almost any version of BASIC. It is modelled after ANSI standard BASIC, and contains many special teaching and debugging commands not implemented by the BASIC that lives in your computer (either Integer or Applesoft). Conversely, both Integer and Applesoft contain many commands and features not implemented by HANDS ON BASIC. Descriptions of how Applesoft differs from the HANDS ON BASIC commands being discussed appear in the margins adjacent to the descriptive paragraph.

To use the HANDS ON BASIC system you'll need an APPLE II series computer with 48K (approximately 48000 "bytes") of internal memory, at least one disk drive (set up for 16 "sectors"), this manual and the HOB System Diskette. We strongly recommend that you also have a printer hooked to your computer, although it isn't necessary. You will also need at least one (and preferably two) initialized diskettes (see the DOS manual that came with your disk drive). Finally, a set of game paddles will be helpful, but not necessary.

This manual has three sections. Section One, COMPUTER THINKING, is a brief introduction to the computer and how it processes information; several of HOB's special system features are also covered. Section Two, HOB TUTORIAL, is a seven chapter tutorial on

the HOB language and system; all the BASIC commands and functions are described, and you are led through a series of carefully designed computer exercises. Extensive use is made of HOB's program tracing displays so that you actually see how the computer executes your programs. Section Three is called PUTTING HOB TO WORK and consists of two chapters on program planning and debugging (including a tutorial on HOB's special debugging tools), plus eleven appendices summarizing HOB's commands, functions, error messages, terms, and so on in a series of charts and glossaries.

Two appendices of particular import are D and J. Appendix D, Errors and Possible Solutions, contains a list of HOB's many error messages with an explanation of what each message means. Each message in the list also has suggestions for what might have caused the error and what action you can take to correct it. Appendix J contains suggested solutions to the problems and practice exercises presented in Chapters 1 through 9. Many of the suggested solutions are either full programs or substantial sections of code, and contain programming techniques not covered elsewhere in the manual.

While you can get a kind of feel for HOB by reading this very friendly manual curled up in your favorite chair, you'll learn infinitely more if you're curled up with your favorite computer. HOB is a learn-by-doing experience. You can't learn programming by reading about it—the only way you can learn how to program is to program.

Each chapter in this tutorial manual is built upon what was learned in the previous ones. There is a constant review built into the sample programs and coding exercises in each chapter so that what you learn in one segment is reinforced in the next. The best way for most people to learn from this manual, then, is to go through its eleven chapters one after the other while sitting at the computer.

We want to encourage you right from the outset to make all the mistakes you can. HOB is forever patient and is specifically designed to help you learn from the mistakes you make. In fact, if you don't make mistakes you don't need HOB. You can take as much time as you like on any programming problem and HOB will wait until you're ready to go on. You can make the same mistake any number of times and HOB will continually show you where you went wrong; good use of its hints and the information provided in this manual will help you understand your mistakes and learn how to correct them—all at your own pace. HOB won't time or chide you—and it doesn't keep score.

Section One:
COMPUTER THINKING

CHAPTER 1: AUTOMATIC ARITHMETIC

Probably the best place to start learning about computers is with something with which you're already familiar: arithmetic. We're not talking about higher mathematics here; you don't have to know much about math to program or to understand a computer (although for some reason people seem to think you do. The author of this manual certainly doesn't!). We're talking about plain old vanilla one-and-one-make-two arithmetic.

In this section you'll have your first hands-on experience with the computer. It is extremely important that you try each of the examples as we go along. Go off on your own if you want; experiment all you like. You can do more than we suggest here, but don't do less! Nobody has ever learned how to program a computer just by reading about it. Don't be afraid to make mistakes; the computer doesn't keep track of them.

Some New Symbols

Most day-to-day arithmetic tasks (dividing up a lunch bill, balancing a check book) call for the use of four basic operations: addition, subtraction, division, multiplication. Most of us have been taught that the operators for these operations look like this:

+	ADDITION
—	SUBTRACTION
÷ or /	DIVISION
×	MULTIPLICATION

Computers use the same symbols for addition and subtraction but the operator for division is always the [/]. The symbol for multiplication is totally different; computers use the asterisk symbol [*] and never the ×. To the computer, × is a variable name (which we'll cover in the next chapter) and is never used as an operator. You'll get used to these minor differences very quickly, and HOB will help you remember.

New Hi-Tech Improved Operators

+	ADDITION
—	SUBTRACTION
/	DIVISION
*	MULTIPLICATION

FIGURE 1

And now, on to the keyboard!



Applesoft BASIC uses the symbol `>` as its ready prompt. However, the Apple's Integer BASIC uses `>` as its prompt. Other languages and computers use different symbols, words, and phrases for their prompts.

The right-facing wedge `[>]` is HOB's symbol for *Ready to accept a command*. The blinking square next to it is called a **cursor** (Latin for *runner* because it runs ahead of what you type. Some people think it got its name from a habit dedicated programmers seem to acquire, but we'll have more to say about that later when we talk about bugs). You can think of the cursor as the tip of an electronic pen that you control through the keyboard.

Type the following into your computer (don't retype the `[>]` symbol. We've included it to help simulate the HOB screen):

`>5+3`

and press the **[RETURN]** key.

When using Applesoft, you must type the word **PRINT** (or its abbreviation, `?`) before the calculation you wish the computer to perform if you want the result to be displayed. The phrase **THE RESULT IS** will not precede the answer: (eg. `PRINT 5 + 3` or `? 5 + 3` instead of `5 + 3`).

Panic Paragraph

If the computer starts flashing warnings and whatnot, don't panic. You've probably just entered some illegal characters and HOB wants you to take them out. Press the `[←]` key a couple of times until the cursor is back to the first position in front of the `>` symbol and start again.

HOB should have responded

THE RESULT IS 8

HOB has assumed you have typed in a problem you would like the computer to solve for you, and has therefore solved it.

Tell HOB to multiply *six times five*. Remember to use the special symbol for multiplication and don't forget to press the [RETURN] key.

You should have typed:

`>6*5 <CR>`

The `<CR>` is computer shorthand for *PRESS THE RETURN KEY*. The word *RETURN* is actually an abbreviation for carriage return, hence the letters CR. You don't type this in.

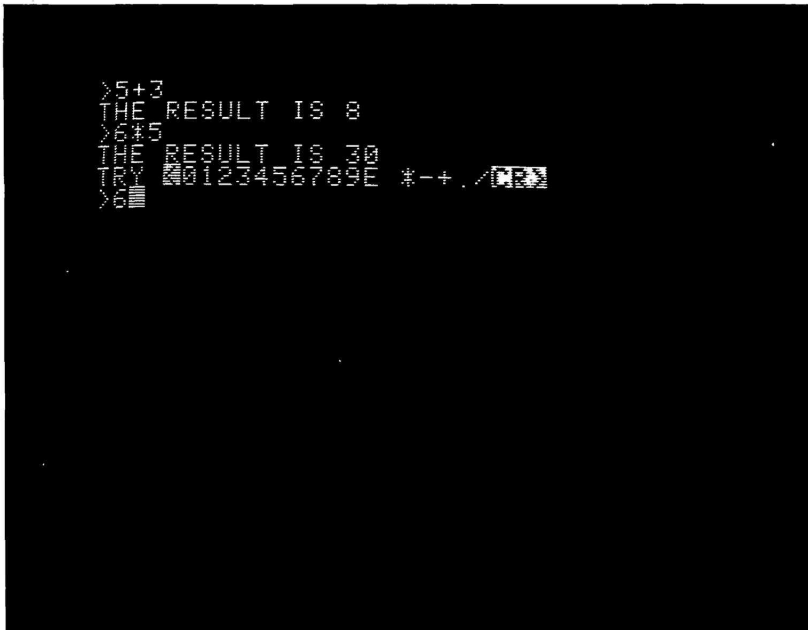


FIGURE 2

If you tried to use the old-style symbol for multiplication or if you tried to include any spaces, you've discovered that HOB won't let you. (If you didn't try it, do so now). There are only so many possible keys that can be pressed after you've entered a number and still have what you've typed be "syntactically correct." If you try to type an illegal character twice in a row, HOB will tell you what keys you can press that will be considered valid. Try it now and see what happens (see FIGURE 2.).

You can enter any number, letter or symbol from the *TRY* list that

you see on the screen, but none other than these. This list of legal characters will change depending on what function you're trying to make HOB perform. It will always appear when you make a syntax error (that is, an error in the way HOB wants to see a command typed) twice in a row.

HOB will never let you get away with syntax errors. But don't worry; it won't tell anybody how many mistakes you make. And you cannot hurt the computer from the keyboard . . . unless you pour a cup of coffee into it.

Back to the work at hand: you can use HOB to do a **chain** of calculations (*a number of calculations in a row*) and you can mix the kinds of operations you do. Thus you can do a series of additions, subtractions, etc. on the same line. For example, you can type in:

>5*6+35-12.5 <CR>

and HOB should say:

THE RESULT IS 52.5

Applesoft has no FINETRACE command. Applesoft does have a command called TRACE which is used to display the sequence of line numbers that is executed when a program is running.

Watching HOB Think

Before we go on, enter this into your computer:

>FINETRACE <CR>

Now re-enter the problem you just gave HOB to solve. This new command will let you watch the order in which HOB solves problems. After you type **FINETRACE**, remember to press the [RETURN] key to get things started. Notice that the first thing HOB did was to work with the multiplication operator. That is, it did the **5*6** part first. Press [RETURN] again and you'll see it do the addition. Finally, it will do the subtraction. HOB does all its operations according to a certain order or *precedence*. Enter this next problem and watch how HOB solves it (the **FINETRACE** command is still in effect):

>125-25*16+4/50-45 <CR>

Remember to keep pressing the [RETURN] key after each step until HOB gives you the final result. What you see in FIGURE 3 should appear on your screen:

HOB went from left to right and solved the multiplication/division parts of the problem first (**25*16** and **4/50**). Then it went back and solved the addition/subtraction parts (**125-400** and so on), again from left to right.

To shut off **FINETRACE**, just enter the command (as you might expect):

>NOFINETRACE <CR>

```

>5+3
THE RESULT IS 8
>6*5
THE RESULT IS 30
>5*6+35-12.5
THE RESULT IS 52.5
>FINETRACE
>5*6+35-12.5
30+35-12.5
65-12.5
52.5
THE RESULT IS 52.5
>125-25*(16+4)/50-45
125-400+4/50-45
125-400+.08-45
-274.92-45
-319.92
THE RESULT IS -319.92
>

```

FIGURE 3

Any time you get an unexpected result issue the **FINETRACE** command and let HOB help you figure out what happened. You might want to leave it in effect for the next section, however; things get a might sticky as we deal with *parentheses*.

Parentheses and Precedence

You can change the way HOB thinks by using *parentheses* characters. Re-enter the numbers and operators from the last problem, but do it this way:

```
>(125-25)*(16+4)/(50-45)<CR>
```

HOB will always solve what's in the parentheses first (again, from left to right), and then go back and solve the problem in its regular order of precedence. You can also *nest* parentheses in a problem (that is, you can have expressions within expressions) up to eight levels deep. In the following example, $12-5$ is nested within $26* \dots -15$ and is said to be the *2nd level of nesting* (mostly by those who talk strangely):

```
>34-(26*(12-5)-15)*(15+12)<CR>
```

The innermost parenthetical expression $(12-5)$ was evaluated

In Applesoft, parentheses may be nested 36 levels deep.

show in numbers (1,000,000,000,000), must be written as 1E12 (1×10^{12}). The actual range of HOB is $\pm 9.99E63$, which if written out numerically would be from minus-to-plus 9.99 times 10 to the 63rd power. This seems adequate to deal with most checkbooks.

Arithmetic Errors

While HOB won't let you make any syntax errors as you enter information, it can't stop you from setting up a situation where an arithmetic error will occur once HOB tries to figure out the problem you've given it. Such errors include dividing by zero or calculating a result too large or too small for the system to handle (causing an overflow condition to occur). Cancel the **FINETRACE** command (**NOFINETRACE**<CR>) and enter the following problem:

$>(3*5 - (7*2))/(30 - (6*4) - (3*2)) <CR>$

ARITHMETIC ERROR 1/0

PRESS <CR> FOR SLOW MOTION REPLAY

Stop! Ignore that computer's suggestion! HOB is about to go through an automatic **FINETRACE** of this problem to show you where the error occurred. Before you take advantage of HOB's proffered aid, see if you can work it through yourself. Using the rules of precedence you've already learned, work the problem out by hand. When you think you've spotted the error, go back to the computer and do as HOB suggests to confirm your own work. Then come back here.

If you were already in **FINETRACE**, HOB wouldn't have gone through the whole process again; it would have stopped trying to solve the problem when it came across the error and would have notified you. The results would be on the screen. You'll get this same sort of error message and help sequence for any arithmetic error. The phrase **ARITHMETIC ERROR 1/0** would be replaced by **ARITHMETIC ERROR exp**, where **exp** will be *the expression that caused the problem*.

Experiment Time

Before going on, take some time to experiment with chained and mixed problems until you have a good feel for HOB's arithmetic. Make as many mistakes as you can. Break the rules—nobody's watching. Consider yourself encouraged to test HOB's limits.

The following are some of the preprogrammed formulae built into HOB. We'll have more to say about functions in Chapter 9; we list these here for the math people who can't wait. A summation of

Applesoft does not always check for syntax errors when a program is typed in. The syntax within a program line is checked when that line is encountered by the computer when the program is run.

all functions in HOB also appears in Appendix C at the back of the manual. In order to operate the following functions, type the function's **NAME** followed by (within parentheses) the number or numeric **VARIABLE** you want acted upon (variables will be covered in detail in the next chapter). For instance, to find the cosine of 12 enter **COS (12)**.

NOTE: HOB is fairly slow calculating certain functions but it is very accurate. patient.

You can also use numeric expressions. **COS(12)**, **COS(4*3)** or **COS(A)** would all yield .84385396, assuming that **A=12**. Exceptions to these rules are clearly noted.

Arithmetic Functions

ABS(exp) ABSOLUTE value of exp (i.e. value of exp without either + or -). **ABS(-12)** will yield 12, **ABS(3*-5)** will yield 15.

EXP(exp) yields the value of the constant e (2.718281 ...) raised to the exp power. **EXP(4)** gives 54.59815, **EXP(-3)** yields .049787068.

INT(exp) the largest INTEGER (whole number) no greater than exp. **INT(3.14159)** will give 3, **INT(-3.42)** will give -4.

LOG(exp) value of the NATURAL LOGARITHM of exp. Value of exp must be positive. **LOG(13)** gives 2.5649494, **LOG(3*.2)** yields -.51082562.

SGN(exp) indication of algebraic SIGN of exp: negative gives -1, positive gives +1, 0 gives 0. Assuming **A=37**, **B=-35** and **C=0** then **SGN(A)** gives +1, **SGN(B)** gives -1, **SGN(C)** gives 0.

SQR(exp) SQUARE ROOT of exp. Value of exp must be positive or "FUNCTION ERROR" message will appear. **SQR(9)** gives 3.

exp1^exp2 exp is RAISED TO THE POWER exp2. **5^3** yields 125, **2^(-5)** yields .03125.

Trigonometric Functions

ATN(exp) yields the angle whose tangent is equal to the value of exp.

COS(exp) yields the COSINE of the angle whose value is equal to exp. Value for exp must be less than 100.

SIN(exp) yields the SINE of the angle whose value is equal to exp.

TAN(exp) yields the TANGENT of the angle whose value is equal to exp.

Values are considered to be **radians** unless the immediate command **DEGREES** is issued.

Values for **SQR** and **LOG** must be *positive*.

AppleSoft's COS function does not have this restriction.

Values for SIN, COS, TAN and ATN are in *radians* unless the command DEGREES is entered. To return to radians enter the command RADIANS.

The *value* for exp in COS(exp) must be *less than 100*; otherwise a FUNCTION ERROR will occur and program execution will stop. Function errors are reported and handled like arithmetic errors. For example:

>(12-3)+LOG(20)*COS(185) <CR>

We'll leave you to the computer and HOB to see what happens.

AppleSoft does not have a DEGREES command; all of its trigonometric functions require radian values.

Chapter Review

Be sure that you understand all of the following new terms, symbols, commands and chart elements before going on. The HOB tutorial is written so that each successive chapter is built on previous ones. You may skip the FUNCTIONS for now if you don't use them in your work or studies.

keys:	[+]	[-]	[/]	[*]	[^]	[E]
	[()]	[RETURN]				
commands:	FINETRACE	NOFINETRACE				
	DEGREES	RADIANS				
terms:	CHAINED	CURSOR	CR			
	MIXED OPERATION	OPERATOR				
	PRECEDENCE	RANGE	RETURN			
	FUNCTIONS	PROMPT				
charts:	FUNCTIONS	OPERATORS	PRECEDENCE			

NOTE: <CR> and [RETURN] both indicate the same key.

Suggested solutions to these problems (and to all others found throughout the manual) can be found in Appendix J at the back of the manual.

Practice Exercises

Solve the following problems by entering them into your computer:

1. Multiply the sum of 6 and 2 by 3 to the 3rd power.

2. Divide the result of 5 minus 3 by one-half.

Solve the following problem on paper first; then use the computer:

3. Predict the result of $5 - 2^3 * (15 - 4) + 83$. Turn on FINETRACE and watch HOB think its way through the problem. Finally, turn off FINETRACE.

CHAPTER 2: COMPUTER VARIABLES

One of the things that gives computers their power is their ability to deal with variables. A *variable* is a symbol that can stand for any one of a number of values. HOB has several kinds of variables: *numeric variables*, used to represent numbers; *string variables*, used to represent words or sentences; and *array variables*, used to represent numbers linked together in some logical pattern. In this chapter we'll deal with numeric and string variables, and save the discussion of arrays for Chapter 7.

Numeric Variables

Enter this into your computer (keeping a space between the word **LET** and the **A**; but as before the space between the number and the **<CR>** is to make things more readable and should not be typed):

```
>LET A=5 <CR>
```

You have just told the computer *Find a place somewhere in your memory banks (I don't care specifically where) to store the value 5. Call that location A.* Note that we don't say *A EQUALS 5*. Rather, we say that *A HOLDS THE VALUE 5*. In shorthand, we can read the expression to say *LET A HOLD 5*. Enter.

```
>A <CR>
```

and HOB will respond with

```
THE RESULT IS 5
```

Now enter

```
>LET A=12 <CR>
```

```
>A <CR>
```

and HOB will tell you

```
THE RESULT IS 12
```

The place in the computer's memory banks that used to hold the value 5 now holds the value 12. The original value is gone and is not recoverable.

We'll define another variable, **B**, and give it an initial value of (or initialize it to) 3:

```
>LET B=3 <CR>
```

In Applesoft, the word **LET** is optional. For example, you could use **A=5** as well as **LET A=5**.

Applesoft allows you to insert any number of spaces between commands.

Variable Arithmetic

Now that we've got two values stored in the computer, we can make it start to earn its electricity. Enter this:

```
>LET C=A+B <CR>
```

Loosely translated, this *statement* tells the computer *Set up a location somewhere in your memory called C and let it hold a value equal to the combined contents of location A and location B.*

```
>C <CR>
```

THE RESULT IS 15

The values of B and A are still intact; we haven't taken the values away from them — we've merely had the computer look at those locations and make a copy of the values held there, add them together, and store the result in a place symbolically called C. Once you've defined a variable it will keep its value until you tell it to change. But don't take our word for it; ask HOB for the values of A and B just to prove it.

Algebraic Thinking Vs. Computer Thinking

If you have a background in algebra you may have drawn certain analogies between the way a computer thinks and the way algebra works. While there are certain similarities between the two, you've got to be careful about how far you carry this analogy. In algebra the following statement would be an impossibility; whereas to a computer, it's as common as applesauce:

```
>LET C=C+1 <CR>
```

In English it says *Take the value stored in location C, add 1 to it, and store the results back in C.* While you'd never find this sort of thing in algebra, it's so common in programming that it has its own name. It's called a *COUNTER*, and you'll be using it later in the HOB tutorial section.

```
>C <CR>
```

THE RESULT IS 16

Once a variable is defined you can treat it as a number in setting up and solving problems. For instance, type in the following *LET* lines:

```
>LET N=35+(15*(6-2)-45) <CR>
```

```
>LET Q=N*12/150 <CR>
```

```
>LET T=A+B+C+N+Q <CR>
```

```
>T <CR>
```

HOB will say:

THE RESULT IS 85

Now type:

>Q<CR>

And this will appear on the screen:

THE RESULT IS 4

Try this now:

>Q+T<CR>

THE RESULT IS 89

The result of function arithmetic can also be stored in variables. Thus you can say:

>LET Q=SQR(81)

>LET N=TAN(C)

>LET C=COS(A+SQR(Q))

That last example warrants closer attention because of the parentheses. HOB will first evaluate the square root of 9 (which is 3). Next it will add that to the value of A (which is 12). Finally HOB will find the cosine of the result, COS(15). If things are still unclear, issue the **FINETRACE** command and type in the last example again; HOB will show you how it processes the problem step by step.

String Variables

Here's where any relationship between algebrese and computerese bites the dust. In algebra, variables are always symbols for numbers or numeric relationships. In a computer, variables can be symbols for words as well. Back to the keyboard:

>LET A\$="ABSOLUTE CONFUSION"

You have now created a *string variable* called A\$. The sign for a string variable is the dollar symbol [\$], usually called *dollar* by those types who like to give names to symbols. The name for any group of characters enclosed in double quotes is called a *string*. In the above example the words **ABSOLUTE CONFUSION** form the *string*.

>A\$<CR>

VALUE IS "ABSOLUTE CONFUSION"

The old value for the numeric variable A is still retained; A and A\$ are totally different variables:

>A<CR>

THE RESULT IS 12

Being highly observant, you've already noted that the *definition of* (setting a value to) the string variable occurred between quotes. The only time when this isn't the case is when you're setting one

string variable equal to another:

```
>LET B$=A$<CR>
>B$<CR>
VALUE IS "ABSOLUTE CONFUSION"

>A$
VALUE IS "ABSOLUTE CONFUSION"
```

HOB Symbols Table

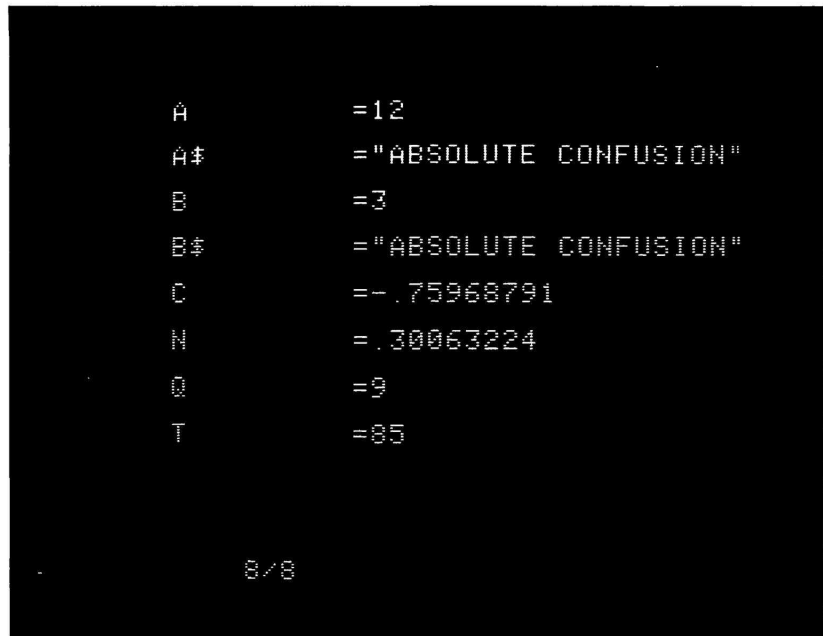
If you're feeling absolutely confused about the values all your variables are holding, there's something you can do about it. HOB maintains a place called **SYMBOLS** where it keeps a list (or table) of all the variables and their current values. You can look at this list the same way you view an individual variable—by typing its name and pressing [RETURN]:

```
>SYMBOLS<CR>
```

If all goes well, the following should appear on your computer screen:

Applesoft has no SYMBOLS command.

FIGURE 5



```

      A      =12
     A$      ="ABSOLUTE CONFUSION"
      B      =3
     B$      ="ABSOLUTE CONFUSION"
      C      =-.75968791
      N      =.30063224
      Q      =9
      T      =85

      8/8
```

The letters are variables and the numbers next to them are the values they hold. The 8/8 to the bottom left means *eight variables*

displayed out of eight variables defined. If you had defined 40 variables, there would be too many to be displayed at the same time on the screen (only eight will fit). The message would then read 8/40 MORE (which would mean *eight variables shown here out of 40 variables defined; there are more that are not being shown yet*). To see the next eight variables you would press the [+] key; continue pressing the [+] to see more. To back up to the previous eight variables shown, you press (what else!) the [-]. At any point you could go back to the first page of variables by pressing the [@] key. Pressing the [RETURN] key terminates the list and the main screen (called the COMMAND screen) will reappear.

Changing The Value of Variables

There are two ways to change the value of a variable. The first is through a LET statement; the second is through entering

>NEW <CR>

This second method is rather radical. Ordinarily used to clear the computer's memory of a no-longer-useful program, it wipes out all variables and makes HOB think you've just turned on the computer. Try it now. Then try to see the SYMBOLS table (chuckle).

Here are the rules for both numeric and string variables. The term *active* in the following charts means *defined for use within a program*.

Rules for HOB Numeric Variables

- There must be a space between the word LET and the name of the variable being defined. There may be no other spaces in the line.
- Any variable that appears to the right of the = sign must have already been defined.
- Variable names may be any *alphabetic* (A-Z) or any alphabetic plus any single *digit* (0-9). Examples are A, Q, N6, J0.

Applesoft numeric and string variable names may be up to 255 characters long, but only the first two characters are recognized by the computer.

Rules for HOB String Variables

- There must be a space between the word LET and the name of the variable being defined.
- Everything to the right of the = sign must be within double quotes (") unless one string variable is being set to another (as in LET A\$=B\$).
- If one string variable is being set to another, then the variable to the right of the = must have already been defined.

NOTE: Each program may contain a total of 63 active variables, string and numeric combined.

An Applesoft program may contain as many variables as the available memory will allow.

Applesoft strings may be up to 255 characters in length.

- A string variable name must be any single alphabetic followed by a \$.
- The string itself (that which is between quotes) may not exceed 18 characters.

Chapter Review

keys:	[=]	[+]	[-]	[@]
commands:	LET	NEW	SYMBOLS	
terms:	ACTIVE	ALPHABETIC	COUNTER	
	DEFINE	DIGIT	DOLLAR	
	INITIALIZE	NUMERIC	STRING	
	VALUE	VARIABLE		
screens:	SYMBOLS	COMMAND		

Practice Exercises

1. Initialize the following variables to the values next to them: A 6; B .2; Q [A minus the product of B times 30]; S [A divided by Q].
2. Have the computer display the values held by each of the above four variables one at a time (HINT: only 2 keystrokes are required for each variable).
3. Have the computer display values for all the variables at the same time (HINT: only ONE command plus a press of the [RETURN] key is required).
4. Add 3 to the variable B (assume you do NOT know the current value of B, but that you DO know that it has been initialized).
5. What's wrong with the following command line?
LET N="SOMETHING'S WRONG"
6. Enter a command that will clear all the variables defined so far.



Section Two:

HOBASIC TUTORIAL



CHAPTER 3: LOOPING AROUND IN HOB

What's in a Program?

A program is a set of coded instructions that makes a computer work. You can think of a program as a collection of numbered cards. Each card is numbered sequentially. The computer looks at the first card, reads the instruction written on it and carries it out. Then it gets the second card and reads its instruction, carries it out, and so on. Occasionally one of the cards will direct the computer to a special subdeck. After carrying out the instructions in the subdeck the program will come back to the main deck and continue following the instructions from the point where it went off to find the subdeck. From time to time an instruction card will tell the computer to skip ahead or to skip back in the deck to a particular numbered card. Whatever the instruction, the computer will follow it precisely.

You've already had some experience issuing commands to your computer. In the last two chapters you were issuing *immediate execution* commands; as soon as you typed a command into the computer and pressed the [RETURN] key the command was carried out. When you write a program and type it into your machine you are issuing a set of *deferred execution* commands that will be carried out later. In HANDS ON BASIC (and in nearly all other BASIC dialects) *deferred commands* are preceded by *line numbers*. The computer looks for the line numbers sequentially so that it can carry out the program's commands in the proper order.

The First Program

Clear out any variables in HOB by issuing the new command in immediate mode. Then try to enter this line of code *exactly as it appears*:

10 PRINT "HI, HUMAN!"

HOB will put this on your screen:



FIGURE 6

HOB's syntax checker is alive and well. As with immediate commands, HOB won't let you make any syntax errors as you're entering deferred commands. The **TRY** line has only two choices in it; a **CR** or a **SPACE**. In this case, you need a **SPACE**:

10 PRINT "HI, HUMAN!"

Did you press the **[RETURN]** key at the end of line? From now on we won't be displaying the **<CR>** symbol; you may have noticed we've already stopped showing the **>** at the beginning of the command line. Just remember that the computer doesn't know what you want it to do until you press **[RETURN]**.

As soon as you pressed the **[SPACE]** bar after you entered the line number, the cursor jumped over a few spaces and the line number was reprinted. This is HOB's way of keeping a tidy screen. Enter this next line of code, but instead of typing the line number (which here happens to be **20**), press the **[SPACE]** bar before you do anything else:

20 GOTO 10

Automatic Line Numbering

There is no automatic line numbering in Applesoft.

HOB thoughtfully typed the line number and the necessary space preceding the command for you. When HOB provides this service, it assumes you want the current line number to be **10** more than the last line. So if the previous line number was **263**, the new number would be **273**. Occasionally you may press the **[SPACE]** bar accidentally when the cursor is in the first position, causing an unintentional automatic line number to appear. Just press the **[←]** to correct the problem and go on typing. Try it now.

The line number range in Applesoft is **0** to **63999**.

Line numbers can be as low as **1** or as high as **9999**. There needn't be any special gap between the line numbers as long as they're in sequential order. Most programmers like to use **10**. You'll soon see why there's a gap left between line numbers at all (**10 . . . 20 . . . 30 . . .**) instead of just numbering them one right after another (**1 . . . 2 . . . 3 . . .**).

HOB allows a code line of up to **32** characters long exclusive of line numbers. Applesoft allows lines of up to **239** characters inclusive of line numbers.

Now that you've entered a couple of lines of code (computer-understandable instructions), you'd probably like to know what your program makes the computer do. Rather than telling you, we'll let the computer show you. You need to enter the immediate execution command **RUN** to make the program work. Type

RUN

(don't forget to press **[RETURN]** and watch the screen).

Dynamic Error Checking

Just as there are syntax rules for entering commands, there are rules governing the construction and operation of programs. HOB wants the last program statement to say **END** so it can know when to stop looking for more lines of code. Accordingly we enter

9999 END

Any line number between 21 and 9999 could be used for this statement. Since we intend to add more lines to this program, we use a high number to insure there will be room (9999 was arbitrary—500 would work just as well). Enter

RUN

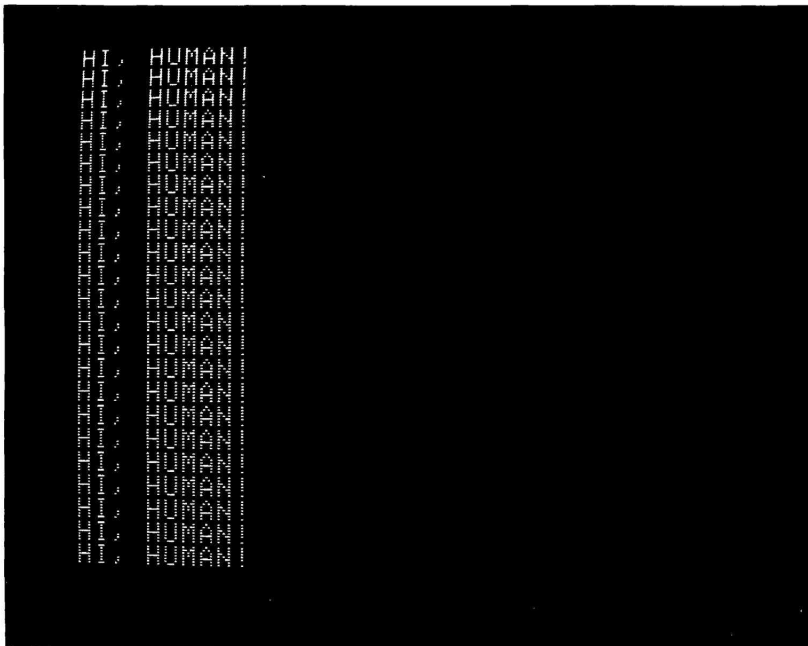


FIGURE 7

Shades of *The Sorcerer's Apprentice*! To stop this madness, press the [ESC] key. That will bring you back to the COMMAND screen.

Incidentally, if there is more than one error in your program, HOB will announce the errors only one at a time. Don't get frustrated if your programs have many errors—it happens to the best of us!

The Infinite Loop

Your first program demonstrates one of Computerdom's favorite *bugs* (that is, errors)—the *infinite loop*. Line 10 tells the computer to

NOTE: To a computer there is a difference between 0 (zero) and O (the 15th letter of the alphabet), as well as between 1 (one) and I (the ninth letter). Don't type letters when you mean numbers or the computer will take what you've typed is a variable!

PRINT on the screen the string "HI, HUMAN!". Having completed that instruction, the computer looks at the next highest line number, line 20, and does what it says. It instructs the computer to look for (line number) 10 and GOTO it. The computer obediently *branches* to line 10, does what it says, and looks for the next highest line number, line 20, and does what line 20 says (which is to go back to line 10). Et cetera, et cetera, et cetera. As we said in the beginning of this chapter, the computer will follow the instructions the program gives it to the letter.

We're going to have to change the program to get out of this mess. We'll use the *counter* technique you read a bit about in the last chapter. Enter this:

```
5 LET X=0
12 LET X=X+1
18 IF X=15 THEN 9999
```

Listing A Program

In order to see your whole program in sequential line order (which is how your computer will look at it) enter the immediate execution command

LIST

FIGURE 8

```

541 FULL
> 10 PRINT "HI, HUMAN!"
> 20 GOTO 10
> 9999 END
> RUN
> 5 LET X=0
> 12 LET X=X+1
> 18 IF X=15 THEN 9999
> LIST
5 LET X=0
10 PRINT "HI, HUMAN!"
12 LET X=X+1
18 IF X=15 THEN 9999
20 GOTO 10
9999 END
> █

```

As you can see, HOB has taken the new program lines and has inserted them into their proper places. A new code line can be inserted into a program any time the program is not in operation (not running) so long as there is a unique line number available. Before going on, see if you can understand from reading the listing what the program you've typed in is doing. Then **RUN** the program:

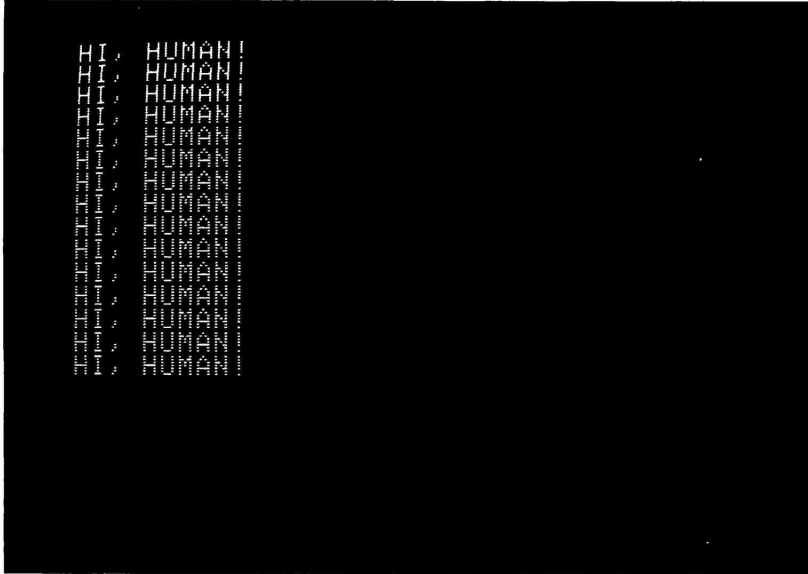


FIGURE 9

Finding The Print Screen

As soon as a program has finished executing, HOB will display the command screen again. The screen displayed above still exists in the computer. HOB calls in the *print* screen, and you can see it again by holding down the [CTRL] key while you press [P]. In computerese, this sequence is called entering a Control P. You don't need to press the [RETURN] key to enter a CONTROL sequence. Try it now. To get back to the command screen, enter a CTRL C (for COMMAND). You can also press [RETURN] to get back to COMMAND, or just start typing anything; HOB will switch you automatically.

Applesoft uses the same screen for displaying commands and the results of a program's PRINT statements. All of the other screens described in this manual are unique to *HANDS ON BASIC*.

How The Program Works

Line 5 initialized (i.e. gave an initial value to) a variable called X. Line 10 prints the message we want. In line 12 we find the *counter*; it *increments* (adds to) the value of the variable X by 1. In essence, X is keeping track of how many times the message is printed onto the

screen. Line 18 checks what value X now holds; if X holds 15 (which in this case would mean that the message has been printed to the screen 15 times) then the program will perform line 9999.

IF/THEN And Conditional Branching

Line 18 is an example of a *conditional branch* because the computer will branch to a different part of the program (instead of going in normal sequential order) only if certain conditions are met. In this case the condition is that the variable X hold the value 15. The form of a *conditional branch* is

IF exp relational exp THEN lnum

where **exp** is any expression, **relational** is one of the relational operators, and **lnum** is a line number.

Applesoft allows any valid command to appear after the THEN in a conditional branch.

Relational Operators

The *relational operators* are symbols that make comparisons between two expressions. There are three relational operators (called simply *relationals* for short), and they can be used separately or in combination. The symbols and their meanings appear in the adjacent column.

=	HOLDS THE SAME VALUE AS
<	IS LESS THAN
>	IS GREATER THAN
<>	DOES NOT HOLD THE SAME VALUE AS
<=	IS LESS THAN OR EQUAL TO
>=	IS GREATER THAN OR EQUAL TO

Some legal IF/THEN statements would be

IF A<B+7 THEN 20

(If A holds less than the value of B plus 7 then GOTO line 20)

IF 2*X>5 THEN 9999

(If the product of 2 times X is greater than 5 then GOTO line 9999)

IF A-B<=Z*12 THEN 50

(If the result of A minus B is not more than the product of Z times 12 then GOTO line 50)

GOTO And The Unconditional Branch

Line 20 is an *unconditional branch* because each time the computer reads this line it will branch to line 10. The command is **GOTO**, and that's what the computer does. You'll sometimes hear this kind of branch called an *absolute branch*; but in this manual we always refer to it as *unconditional*.

The "Major Program Loop"

Notice that lines 5 and 9999 are executed only once in this program, while lines 10, 12, 18 and 20 are executed a number of times

and make up the *major program loop* (the computer LOOPS around these lines to do the major work of the program).

By the way, three of these four lines are executed 15 times and one of them is executed only 14. Budding Genius Award goes to the person who can say which line is executed only 14 times and why (you can find the answer in Appendix J at the back of this manual. But don't look until you try to figure it out yourself).

As an experiment, change line 20. Line numbers must be unique; if we type a new line 20, the old one will be destroyed. So it goes:

```
20 GOTO 5
RUN
```

We've seen this before! Press [ESC] to get back to the command screen ([ESC] will always interrupt a running program and [RETURN] will make the program continue from where it left off. Try it!). We've set up another *infinite loop* here. When the computer executes line 12, X gets incremented just as it should; but the value of X never reaches 15. Line 20 sends the computer back to line 5 for the next instruction, and line 5 resets the counter to zero.

To demonstrate another bug (to make sure you recognize it when it occurs in a more serious program) change line 20 again. This time, tell the computer to branch to line 500. Then RUN the program:

```

        60 FULL
END
5 LET X=0
10 PRINT "HI, HUMAN!"
12 LET X=X+1
18 IF X=15 THEN 9999
20 GOTO 500
9999 END

>LIST

5 LET X=0
10 PRINT "HI, HUMAN!"
12 LET X=X+1
18 IF X=15 THEN 9999
20 GOTO 500
9999 END

>RUN

20 GOTO 500
500 NOT FOUND

>

```

FIGURE 10

The error statement that Applesoft will give you is UNDEFINED STATEMENT ERROR IN LINE 20.

The number indicated is a *target* line that doesn't exist. Since the line isn't there, the computer can't look there for its next instruction. You can't say it didn't try, though.

Using the Disk Drive

Change line 20 back to the original GOTO 10. Next *list* the program so that you can make sure it's intact and that you've cleared up all the bugs we introduced (Figure 3-F shows what the listing should say). Then save your program to the disk drive using the **ROLLOUT** command. **ROLLOUT** is a disk command instructing the computer to store a named program for later use. We've named our program **FIRST**, but you can call it whatever you like, as long as you follow these rules:

Apple's DOS 3.3 uses the command **SAVE** instead of **ROLLOUT**.

Apple's DOS 3.3 allows file names to contain up to 30 characters, and special characters may be used in the name after the first character.

- The name must have from 1 to 27 characters
- The first character must be a letter
- Other characters may be numbers or letters

Rollout

The **ROLLOUT** command has the form

ROLLOUT <PROGRAM NAME>

where **<PROGRAM NAME>** is whatever you come up with conforming to the above rules. Your screen should look like this:

FIGURE 11

```
> 5 LET X=0
> 10 PRINT "HI, HUMAN!"
> 15 LET X=X+1
> 18 IF X=15 THEN 9999
> 20 GOTO 500
> 9999 END
> RUN

      20 GOTO 500
      500 NOT FOUND

> EDIT 20
> 20 GOTO 10
> ROLLOUT FIRST
PLS PLACE ROLLIN-ROLLOUT DISK IN DRIVE
PRESS SPACE TO CONTINUE
```

The ROLLIN-ROLLOUT diskette is any *initialized* diskette you've chosen to hold your completed programs. (For information on initializing diskettes, see the DOS manual that came with your diskette drive. Read the pages of that manual on the proper handling of diskettes). Be sure that you have your ROLLIN-ROLLOUT diskette properly inserted in the drive before you press [RETURN] (see the introductory section to this manual for instructions on inserting diskettes).

After you've **ROLledOUT** your program and the cursor comes back you're free to experiment, now that the program you've written is safely stored away. No matter what you do to the program in memory, the program you've stored out to the diskette will not change.

CAUTION: If you **ROLLOUT** another program with the same name as one already stored on the diskette, the original stored program will be destroyed.

Catalog

To make sure you don't try to store two programs under the same name, you had best check the names of programs already on the diskette. HOB allows you to see what's on the diskette by entering the command

CATALOG

Do as the screen directs, and you'll see displayed on the screen a list of everything on the diskette.

Each stored program is listed with three pieces of information. The letter to the far left is the program's *FILE TYPE*, about which you needn't concern yourself (or see the DOS manual). The three-digit number next to it is the program's *SIZE*, measured in *SECTORS* of 256 BYTES (again, see the DOS manual for more information). And finally comes the program's *NAME*.

Two-Drive Systems

If you're fortunate enough to have two disk drives, you can save yourself some time by inserting your ROLLIN-ROLLOUT diskette into Drive #2. When you save the program, use the form

ROLLOUT <PROGRAM NAME>, D2

Since you have diskettes in two drives, you have two separate **CATALOG** lists; as you might expect, to see a list of programs stored on the diskette in the second drive enter the command

CATALOG, D2

And to see what's in the first drive again, enter

CATALOG, D1

It's somewhat inaccurate to say that the program was stored to

diskette; actually only a copy of the program was stored. The program is still in the computer. **RUN** it to make sure; then clear the program memory (that is, erase the program and variables and everything else you've entered so far) by entering the **new** command.

Rollin

The screen should have cleared, and entering the command **LIST** shouldn't cause much to happen. Now enter the command

ROLLIN <PROGRAM NAME>

where **<PROGRAM NAME>** is whatever name you gave to your program. In our case, we're entering

ROLLIN FIRST

The disk drive will whirr again. When it stops, not only will you have a copy of your stored program back in the computer, but all the information that was on the command screen when you saved the program will be back as well! And a copy of the program (and all pertinent information) is still safely stored on the diskette.

Ain't technology grand?

Deleting Programs

Diskettes can store several programs, the exact number depending on the size of the programs. Eventually you're going to run out of space on the diskette (HOB will tell you when that happens). You may use another initialized diskette to store programs, or you may reuse the same diskette. In order to reuse the same diskette, you'll need to make room by removing a program already stored. The command to do that is

DELETE <program name>

To see this command in action, send the current program in memory out to the diskette under a different name than the one you've already used. We're using the name **TEST**. Then **CATALOG** the diskette. There should be three programs listed: the diskette's initializing program (ours is called **EDU-WARE**, see the DOS manual) plus the two programs you've **ROLLED OUT**. Ours has **FIRST** and **TEST**. Now **DELETE** the program **TEST**, and issue the **CATALOG** command again. There should be only one name left on the diskette, that of the original program. The series of commands we used was:

ROLLOUT TEST
CATALOG
DELETE TEST
CATALOG

Apple's DOS 3.3 uses the command **LOAD** instead of **ROLLIN**.

WARNING: Once a program has been deleted from the diskette, it cannot be recovered. Make sure you really want to wipe out a program from the diskette before deleting it. **DELETE** is to a program on diskette what **NEW** is to a program in memory.

Preventing Accidental Deletions

HOB has a disk command to help prevent accidental disk **DELET**Es, the **LOCK** command. **LOCK** makes a program **unDELETE**able until you **UNLOCK** it. The command has the form

LOCK <program name>

You can recognize programs that are **LOCK**ed by the asterisk (*) that appears in the extreme left column near the program name when you **CATALOG** a diskette.

To make a program **DELETE**able, issue the command

UNLOCK <program name>

Before going on to the next chapter be sure you understand all the commands and terms listed below. Make sure you know the uses of the new keys listed. We recommend strongly that you do the practice exercises as a kind of self-test (suggested solutions are in Appendix J).

Chapter Review

Keys:	[←]	[ESC]	[CTRL]	[SPACE] bar
	[=]	[<]	[>]	D
Commands:	END	GOTO	RUN	LIST
	PRINT	CTRL C	IF/THEN	
	ROLLIN	ROLLOUT	CTRL P	
	CATALOG	DELETE	LOCK	UNLOCK
Terms:	BRANCH	DEFERRED	EXECUTION	
	IMMEDIATE	INITIALIZE DISK		
	LINE NUMBER	LISTING	LOOP	
	PROGRAM	TARGET ERROR		
	RELATIONAL			
Screens:	PRINT			
Shorthand:	<i>exp</i>	<i>lnum</i>		

Practice Exercises

1. Write a program that will PRINT the word "HERE" 10 times, then the word "THERE" once. Use a COUNTER and an IF-THEN command to do it.
2. Write some code that will count from 1 to 10 on the screen. You can reuse most of the lines from the program you wrote for #1 to do it.
3. What's wrong with this program?
10 LET Q=0
20 LET Q=Q+1
30 IF Q=5 THEN 50
40 GOTO 10
50 END
4. Put into words line 30 of the above program.
5. What's the difference between line 30 and line 40 of #3's program?
6. What's the difference between the commands RUN and LIST?
7. How would you go about storing the program in #3 onto a diskette? How could you get it back into the computer?

CHAPTER 4: LIMITED LOOPING

In the last chapter you were introduced to the LOOP. The loop was controlled by an IF . . . THEN statement; if a certain numeric variable held a certain value, the computer was to EXIT (branch out of) the loop. In this chapter we'll deal with two more ways to control looping—the FORLOOP and the DOLOOP.

Forloops

Enter this program:

```
10 FOR L=1 TO 20
20 PRINT L
30 NEXT L
999 END
RUN
```

Now call the PRINT screen back [CTRL][P] and notice the first and the last numbers displayed. They're the same numbers as appear in line 10. Use [CTRL][C] to call the COMMAND screen and rewrite line 10 to say:

```
10 FOR L=6 TO 15
```

RUN the program again. See if you can figure out what's happening before reading the explanation.

HINT: think of line 10 as saying *LET THERE BE A RANGE OF VALUES FOR L GOING FROM 6 TO 15*; think of line 30 as meaning *SET THE NEXT VALUE FOR L AND GOTO WHATEVER LINE FOLLOWS THE "FOR" LINE*. If the two different line 10's on the screen are confusing LIST the program; LIST always gives you the version of your program with your most recent line changes.

The form of this command pair (the two commands work together) is

```
FOR index=lowlimit TO highlimit
•
•
•
NEXT index
```

where **index** is any numeric variable (the only kind you don't have to initialize) and **lowlimit** is the beginning number or numeric variable,

and **highlimit** is the ending number or numeric variable. The **FOR** statement is the first line in the loop and the **NEXT** statement is the last line. Any number of lines can come in between the **FOR** statement and the **NEXT** statement to make up the **BODY** of the loop (those commands that get repeated). In our example there's only one line in the loop's body, line 20.

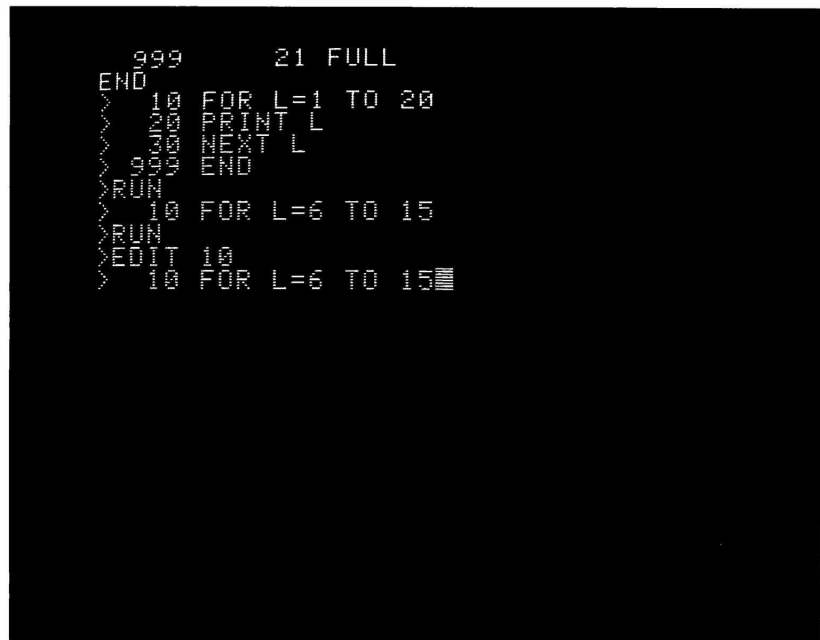
In the original program, **lowlimit** was 1 and **highlimit** was 20 so the numbers 1 to 20 were printed (by line 20). The first time through the loop, L held the value 1; the second time it held 2 and so on up to 20. After you changed line 20 and modified **lowlimit** and **highlimit**, the range of values for L was also changed; the values went from 6 to 15. Accordingly the numbers 6 through 15 were printed.

Editing

We'll make another change to line 10 to further demonstrate **FOR** ... **NEXT**, but rather than retyping the *whole* line we'll use HOB's built-in facilities to *edit* it. Enter the command

EDIT 10

FIGURE 12



```

999      21 FULL
END
10  FOR L=1 TO 20
20  PRINT L
30  NEXT L
40  END
> RUN
10  FOR L=6 TO 15
> RUN
> EDIT 10
10  FOR L=6 TO 15

```

Applesoft does not have an **EDIT** command. However, [ESC] [I], [J], [K], and [M] may be used to reposition the cursor, and the [→] key may be used to copy the line.

HOB has retyped line 10 and has placed the cursor at the end. From here we could go backwards in the line by using the [←] or we

could add to the line — which is what we want to do. Add a space and the words **STEP 2** so that the line looks like

10 FOR L=6 TO 15 STEP 2

and **RUN** the program.

The print screen went by pretty quickly; call it back with a **[CTRL][P]** and see what happened. Instead of the numbers increasing in single steps (which is what **FOR . . . NEXT** will do automatically unless you tell it otherwise) they've gone by 2's (that is, the value of the **index** variable **L** has increased by 2). The highest number displayed on the print screen is 14 because that's as high as **FOR . . . NEXT** could go without violating **highlimit** (in this case, 15) by going past it.

FOR . . . NEXT can also step backwards. Rewrite line 10 to say

10 FOR L=15 TO 6 STEP -1

and **RUN** it again. Now try it by half-steps; using **EDIT**, rewrite the line to end in **STEP -.5** instead of **STEP -1**.

Label that Variable!

Instead of just having numbers run down the computer screen we might as well have the computer tell us what the numbers mean. Add line 15 to the program (which will also add a new command to your HOB knowledge):

15 PRINT "THE VALUE FOR L IS";

Up to now all **PRINT** statements have caused *text* (printed numbers and letters) to appear on separate lines on the computer's screen. That **SEMICOLON (;)** at the end of the line tells the computer: *Take the text from the next PRINT statement and make it show up on the same screen line as was just displayed.* **RUN** the program to see what all this is about.

Actually with the proper editing we can combine two lines of code. Using a single print statement with the semicolon command you can display a series of numeric and/or string variables on the same screen line. First **DELETE** (remove from the program) line 20 by typing its number and pressing **[RETURN]**. HOB will respond with

20 DELETED

(You could have also entered **DEL 20** and had the same results). Now add the numeric variable **L** to the end of line 15 by using the **EDIT** command. Then **RUN** the program.

Usually on the Apple, [ESC] [I] is used to move the cursor up, while [CTRL] [O] has no special function.

... and More Editing

HOB has edit commands that allow you to insert and delete characters in the middle of a line. We'll use line 15 to experiment. We want it to end up looking like this:

```
15 PRINT "THE VALUE FOR VAR L IS";L
```

Here's how to get there by using some new commands. **EDIT** line 15. Using the [←], back the cursor up so that it's over the first L (the one in the string). Now press [CTRL][I] six times (I for INSERT). Each [CTRL][I] will insert one blank space and move every character to the right one space. Type the word **VAR** (an abbreviation for *VARIABLE*). Close up the gap between the semicolon and the L by pressing [CTRL][O] (for *OMIT*) until the cursor is one space to the left of the L. Use the [→] to move the cursor to the end of the line and press [RETURN]. If you become hopelessly entangled in your own editing and introduce more mistakes than you can change, enter [CTRL][X] to cancel the line; then either re-edit or just retype the whole line. Here's a summary of HOB's editing commands; a similar summary is in Appendix H at the back of the manual:

Editing Commands

EDIT <i>Inum</i>	prepare to EDIT existing line
[←]	move cursor over character to left
[→]	move cursor over character to right
[CTRL][I]	insert blank space at cursor; move characters to right of cursor one place to right (I for INSERT)
[CTRL][O]	delete character under cursor; move characters to right of cursor one place to left (O for OMIT)
[CTRL][X]	cancel line being typed; does NOT change line in memory
DEL <i>Inum</i>	remove line <i>Inum</i> for program
<i>Inum</i>	same as DEL <i>Inum</i>
DEL <i>Inum1, Inum2</i>	remove all lines between <i>Inum1</i> and <i>Inum2</i>

You can use all these commands (except EDIT) when you're entering program lines for the first time. Use EDIT on lines that are already part of a program.

Here's what your program should look like after you've done all that editing to line 15:

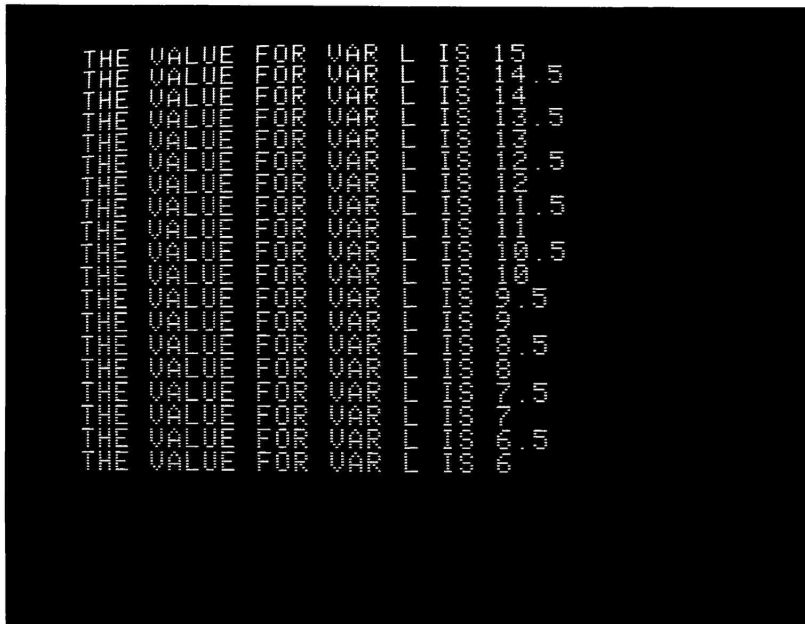
```
10 FOR L=15 TO 6 STEP -.5
15 PRINT "THE VALUE FOR VAR L IS";L
```

999 END

Now **RUN** the program. Figure 4-B shows you what your display screen should look like.

A Programming Problem . . .

We now know a bunch of values for `L`. But what we don't know is how many times the computer has gone through the loop. Using three lines of code you can initialize a variable to be used as a counter (before the loop), set up the counter (inside the loop), and have the computer tell us the total times the machine looped (after the loop). Do this and then come back here and compare your method to ours.

**FIGURE 13**

... and a Suggested Solution

The solution that we're using here isn't the only possible solution. If your solution worked then your solution is right! We solved the problem by adding these lines:

```
5 LET C1=0
12 LET C1=C1+1
40 PRINT "THE LOOP COUNT WAS";C1
```


And our whole program looks like this:

```
5 LET C1=0
10 FOR L=15 TO 6 STEP -.5
12 LET C1=C1+1
15 PRINT "THE VALUE FOR VAR L IS";L
30 NEXT L
40 PRINT "THE LOOP COUNT WAS";C1
999 END
```

Delay Loops: Stalling for Time

We've been talking about what comes in the loop's **BODY**, that which gets repeated between the **FOR** and the **NEXT** statements. Sometimes the loop has no body; the loop is empty. In effect, you set up a situation where the computer just sits there spinning its wheels (or electrons, computers don't have wheels). Sometimes that can be quite useful, especially when you need a time delay between events.

By the merest coincidence we could use such a *DELAY LOOP* in the program we've been writing. We don't have a chance to read what's on the screen before the program ends and the command screen reappears. Add these two lines to the most recent program and **RUN** it:

```
21 FOR Z=1 TO 50
22 NEXT Z
```

Change the value of **highlimit** in the delay loop and see how it changes the delay time between displayed messages on the screen.

HOB's Forloop Tracking Screen

RUN the program again, but as soon as you press the **[RETURN]** key press the **[F]** key. What you see is HOB's *FORLOOP* screen (reproduced in figure 4-C). It keeps track of all the forloops currently being acted upon by the computer. As the program runs, pertinent information about your forloops is displayed here. While you look at this screen you can't see what's being displayed on the print screen (press **[P]** to do that and then press **[F]** to get back here); but you already know what's being shown there.

HOB has a number of these *PROGRAM TRACKING SCREENS*; their purpose is to let you see valuable information about your programs while they operate. You'll be introduced to each of these screens as we continue. You can find a synopsis of all the *PROGRAM TRACKING SCREENS* in Appendix E.

For each active loop, the *FORLOOP* program tracking screen gives the **FOR** code line, the **NEXT** code line, and information about the loop's values: *F* is the *first value* for the index, *L* is the *last value*, *I* is the current *index value* and *S* is the *step value*.

```

22      229 FULL
-----
10 FOR L=15 TO 6 STEP -.5
F=15      L=6
S=-.5     I=13
30 NEXT L
-----
21 FOR Z=1 TO 50
F=1      L=50
S=1      I=9
22 NEXT Z

```

FIGURE 14

Here we have another one of those *nested* situations; one **FOR...NEXT** loop is nested within another. The **Z** loop happens within the **L** loop. The value of **L** doesn't change until the **Z** loop is completely finished. That's the way it is with *FORLOOPS*; you can nest them (in combination with *DOLOOPS*, which we'll get to in a few paragraphs) up to four levels deep, but the innermost one must be finished before the next outermost one can continue. If, for instance, you had made a mistake entering line 22 and had typed 32 as the line number, you would get an *INDEX DOES NOT MATCH* error. Try it. Then correct the error.

Applesoft allows you to nest **FORLOOPS** up to 10 levels deep.

Applesoft error message will be **NEXT WITHOUT FOR** error.

The Query Key

RUN the program again; as soon as the program begins, press **[F]**. The number at the top left of the screen tells you what program line is about to be executed, the number next to it tells you how many commands have been executed since the command **RUN** was entered, and the word **FULL** tells you about the program's *speed of execution* (about which we'll have more to say in the next chapter). Press the **[Q]** key now (**QUERY**) and information will appear on the right side of the top of the screen.

The letter F at the far right tells you you're watching the *FORLOOP* screen. The letter L five spaces to the left tells you that at least one loop is active; the number next to the L tells you the number of the innermost loop currently being executed. The photograph in figure 4-C was taken when the program was at line 21. That's the first line of our new delay loop. The number at the upper right refers to this loop since it's the innermost active one. These letters and numbers (along with other numbers we'll describe as we come to them) will appear at the top of all *PROGRAM TRACKING SCREENs* when you press the [Q] key. They'll disappear when you press the [Q] key again.

If the program is still running, press the [ESC] key to interrupt it. Now look at the print screen by pressing [CTRL][P]. As we pointed out in Chapter 3, when a program is stopped for any reason you can still view any *PROGRAM TRACKING SCREEN* by using its ordinary calling key entered as a CONTROL character. Notice what happens when you change a program in any way — which you're about to do.

Finish off this program by adding another delay loop at lines 51 and 52 so that the message printed by line 40 won't flash by so fast:

```
51 FOR Z=1 TO 50
52 NEXT Z
```

Now try to call the *FORLOOP* screen by pressing [CTRL][F]. HOB won't let you! That's because the information on the various *PROGRAM TRACKING SCREENs* is no longer valid. Changing any code in a program makes it a whole new program; most old information is no longer valid. To access the screens now, you'll have to **RUN** the program again so that new valid information can be generated.

RUN the program and change the value of **highlimit** until you feel comfortable with the delay. Then save the program to disk (we're saving our version of the program under the name **FORLOP**). Remember: the command is

ROLLOUT <PROGRAM NAME>

RUN the program again while watching the *FORLOOP* screen. Information about the delay loop at lines 51 and 52 is displayed on its own; all the data about the L and the first Z loops has disappeared. The only active loop left in the program is this final delay loop.

Clear your computer by entering **NEW** and we'll look at the final type of loop we'll cover in HOB — the DO loop.

REASON: These loops are no longer active.

DO, DO ... UNTIL, DO ... WHILE, and LOOP are not implemented in Applesoft.

DOLOOPS come in three varieties, depending on what you want them to do. The first type sets up what essentially is an *infinite loop*.

Doloops

Enter these lines:

```

5 LET N=0
10 DO
30 LET N=N+1
35 PRINT N
40 LOOP
999 END

```

Now **RUN** the program. When you get bored watching your computer display numbers on the screen, [ESC]ape out of the program. Here's what the program would say if it were written in English:

5	<i>Initialize a variable N to zero</i>
10	<i>Start the loop here</i>
30	<i>Increase the value of N by one</i>
35	<i>Print the current value of N</i>
40	<i>Go back to the start of the loop</i>
999	<i>End the program</i>

The *DO* in line 10 starts the loop; the *LOOP* in line 40 sends the computer back to the *DO* in line 10. Line 999 never gets executed; there's no way to end the program. This structure is useful if you have a program you want to run for an indefinite time period.

Do Until

This *DO UNTIL DOLOOP* combines *DO* with a *RELATIONAL* statement. Its form is

DO UNTIL exp relational exp

where **exp** is any expression and **relational** is any of the *relational operators*.

Delete line 32 and change line 10 to:

```

10 DO UNTIL N=20

```

Your program should now say:

```

5 LET N=0
10 DO UNTIL N=20
30 LET N=N+1
35 PRINT N
40 LOOP
999 END

```

Line 10 says *repeat the following loop until N has the value 20; then branch to the first command line following the LOOP command*. Here's another way to do the same thing, using the final type of *DOLOOP*.

Do While

DO WHILE takes the form

DO WHILE exp relational exp

Again, change line 10:

10 DO WHILE N<20

This time, line 20 says *repeat the following loop while N has a value less than 20; then etc.* Once again, the effect is the same.

DOLOOPS may be nested to four levels, either by themselves or in combination with *FORLOOPS*.

Do On The Forloop Screen

When *DOLOOPS* appear on the *FORLOOP PROGRAM TRACKING SCREEN*, the command lines for both the *DO* command (of whatever type) and the *LOOP* command appear; between them appears one blank line, plus (in the case of the *DO WHILE* and *DO UNTIL* types) one line displaying the current values of the two expressions compared by the relational operator. **RUN** the most recent version of the program watching the *FORLOOP SCREEN* for a demonstration. Figure 4-D shows you what your screen should look like.

Now spend some time playing with *FORLOOPS* and *DOLOOPS*. Experiment. Take the program in memory and change it around. Make mistakes.

FIGURE 15



```
40      48 STEP      █
-----
10 DO WHILE N<20
11<20
40 LOOP
```

Chapter Review

Keys: [→] [;] [F] [Q]

Commands: FOR-TO-STEP & NEXT EDIT
DO & LOOP DO-WHILE & LOOP
DO-UNTIL & LOOP DEL & *Inum*
CTRL O CTRL I CTRL X
CTRL Q IF-THEN EXIT IF-THEN NEXT

Terms: BODY DELAY LOOP DELETE
DOLOOP QUERY EDIT
FORLOOP TEXT
PROGRAM TRACKING SCREEN

Screens: FORLOOP

Charts: EDITING

Shorthand: *index* *highlimit* *lowlimit*

Practice Exercises

1. Write a program 4 lines long that will display the numbers 6 through 12 on the same line.
2. Use EDIT to change one line in the program you just wrote so that the displayed numbers will be 12 down through 6.
3. Write a program that will add the numbers 1 through 50 (as in $1+2+3 \dots +50$) and will display the results (you may be faced with a blank screen for 15 seconds or so while the program works). It shouldn't take more than 10 lines; it can be done in 6 lines.
4. Modify the above program so that, as it is executing, the number being added is displayed in one column while the running total is displayed in another. It can be done by adding just one line. The display should look like this:

1	1
2	3
3	6
4	10
•	•
•	•

CHAPTER 5: THE READ/DATA/RESTORE COMMAND SET

Back in the first chapter you played with the computer as if it were a calculator. You typed in a couple of numbers with an operator in between and the computer either added or subtracted or something-else the numbers. The computer was operating in IMMEDIATE mode. Of course, the computer can do the same things it did there in DEFERRED mode too.

Enter this into your computer and RUN it:

```
10 LET A=10
20 LET B=20
30 LET S=A+B
40 PRINT A;" + ";B;" IS";S
50 FOR X=1 TO 100
60 NEXT X
999 END
```

The semicolons in line 40 keep the display all on the same line. Lines 50 and 60 are just a little delay loop to keep the PRINT screen from disappearing too fast.

Here's the computer using the same technique to add a bunch of numbers:

```
10 LET A=10
20 LET B=15
30 LET C=36
40 LET D=123
50 LET E=47
60 LET F=12
70 LET G=15
80 LET H=13
90 LET I=19
100 LET S=A+B+C+D+E+F+G+H+I
110 PRINT "THE NUMBERS' SUM IS";S
120 FOR X=1 TO 100
130 NEXT X
999 END
RUN
```

Assigning Values Via READ/DATA

Now here's a different way to do the same thing, making use of a new command pair: **READ** and **DATA**. This time, instead of assigning the values to variables, all the different values will be listed in the same line (line 20):

```
10 LET S=0
20 DATA 10, 15, 36, 123, 47, 12, 15, 13, 19
30 FOR X= 1 TO 9
40 READ A
50 LET S=S+A
60 NEXT X
DEL 70,100
RUN
```

In the next three paragraphs we're going to explain how the program works. Then we'll introduce you to another HOB feature that will let you see in super-slow motion how the **CODE** (another word for programmed instructions) gets executed.

LIST the program before you read any further so you can follow along as we describe the program flow, line by line. In line 10 the variable **S** is *initialized*; it will later be used to keep a running **SUM** of numbers added together. Line 20 gives the computer **DATA** to act upon (what good's a data processing machine without any data to process?); the "acting upon" happens in lines 30-60. These lines make up the major program loop. Line 30 starts the **FORLOOP** with a **highlimit** of 9, the number of items in the **DATA** list. Line 40 instructs the computer to **READ** (get a value for) **A**; that value will come from the **DATA** list. Line 50 we've seen before. In earlier chapters we saw this construct used as a **COUNTER** with the number + 1 added to it a number of times. Here the value of **A** is added to **S**.

After the computer has executed line 50, the variable **S** will hold the value 10. Line 60 pushes the value of the **FORLOOP** index up by one, and the computer branches back to line 40 to get another value for **A** from the **DATA** list. The computer keeps track of what items in the list have been **READ** (it reads them in order from the lowest line numbers to the highest and from the leftmost item in a line to the rightmost); since it's already used up the 10 it assigns the value 15 to **A**. Line 15 adds that new value to what was already in **S**. **S** held 10; 10 + 15 equals 25, so now **S** holds 25. And so it continues until the **DATA** list is exhausted (nine **READ**ings). The program concludes by displaying the sum of the numbers: the final value for **S**.

The **DEL** command eliminated lines 70 through 100; they were left over from the previous program. Also left from the previous program were lines 110 through 999 but we wanted to recycle them. We replaced the original lines 10 through 60. We kept what was useful from the old program and threw the rest away; it's a trick experienced programmers use to save needless typing.

The Chronological Trace Screen

Let's have the computer show us how this stuff works. **RUN** the program again, but this time as soon as you type **RUN** and press the **[RETURN]** key, press the **[SPACE]** bar. Pressing the **[SPACE]** bar any time a program is running will suspend the program where it is without returning to the **COMMAND** screen. To the computer, time has stopped and everything is frozen (who needs H. G. Wells?). Now press the **[T]** key. You are looking at another of HOB's *PROGRAM TRACKING SCREENS*. This one is called the *CHRONOLOGICAL TRACE SCREEN* (or *CTRACE* for short) since it shows each line as it is executed in chronological order.

Depending on how quickly you pressed the **[SPACE]** bar your screen should look something like figure 16.

If this much information isn't displayed on your screen, press the **[SPACE]** bar. Each time you press the **[SPACE]** bar another line of information will appear. Keep pressing it until your screen matches the one shown in figure 16. Each time you press the **[SPACE]** bar the computer executes one more program line. The number at the top left of the screen indicates what line the computer will execute next. We're actually watching the computer **RUN** this program one line at a time.

You can see the order in which the lines of the program are executed. You can also see each time a variable changes its value. Variable **S** for instance, first set in line 10, changes again in line 50. The index for the **X FORLOOP** changes in line 60. Speaking of the **FORLOOP**, if you press the **[F]** key you can see the condition of the **FORLOOP**; after that you can press the **[T]** to rejoin the rest of us.

If you press the **[SPACE]** bar now you'll see **A** assigned the value 15. Press it again and you'll see **S** increase in value to 25. Keep pressing the **[SPACE]** bar; lines 40, 50 and 60 keep recurring (they're the body of the **X FORLOOP**). Finally when the value for **X** exceeds **highlimit**, the **FORLOOP** is exited. After line 110 appears you'll notice a flashing 0 at the top of the screen. This is HOB's way of telling you that

```

50      7 STEP

10 LET S=0
S=0
20 DATA 10,15,36,123,47,12,15,13,19
30 FOR X=1 TO 9
X=1
40 READ A
A=10
50 LET S=S+A
S=10
60 NEXT X
X=2
40 READ A
A=15
50 LET S=S+A

```

FIGURE 16

something is on the PRINT screen. Press [P] to see it; then press [T] to come back here.

You can see that the 0 has stopped flashing. Once you've responded to HOB's PRINT SCREEN ON notice, the notice goes away. It just wanted a little attention.

HOB's Time Machine

HOB allows you to control the speed at which a program will execute in other ways. While the [SPACE] bar will make the program single step, the [RETURN] key will make the program run at full speed again. The [←] and [→] (which up to now you've been using to EDIT program lines) can be used to slow down and speed up program time. Assuming the program starts out running at full speed, pressing the [←] key once will put in a 1/8 second execution delay between lines. A second press puts in a 1/4 second delay. A third press will make the program pause 1/2 second between program lines, a fourth introduces a full-second pause, and a fifth stops the program in its tracks (Come to think of it, a fifth will stop most of us in our tracks). Pressing the [→] key speeds up the program a notch at a time until it's running at full speed (see the chart at the end of the next section).

X—The “Full Speed Shortcut”

Quite often you’ll want to get the program running at full speed and then switch to the PRINT screen to watch the progress of displayed messages. For those situations, just press [X] for (eXPRESS) and HOB will take care of the rest. Of course, [X] only works during program execution.

Pace

HOB also has a command called **PACE** to be administered in immediate execution mode only. The form is

PACE=n

where **n** is a whole number between 0 and 5. At 0 a program is in single step mode; at 5 it runs at full speed. **PACE=1** sets program speed to approximately one line per second. The [→] and [←] keys take precedence over **PACE**; [RETURN] during execution sets **PACE** to 5. **PACE** is also reset to 5 by [CTRL][X] during execution or by entering the **RUN** command while the program is stopped. HOB also allows you to say

PACE=FULL

to mean full speed, and

PACE=STEP

to mean single step. In case you’re confused by all this speeding around, here’s a chart to make things easier:

SPEED	SET PACE TO	PRESS [←]
full	5 or FULL	—
1/8 sec delay	4	once
1/4 sec delay	3	twice
1/2 sec delay	2	thrice
1 sec delay	1	four times
single step	0 or STEP	five times

Meanwhile, at the top center of all PROGRAM TRACKING SCREENS (except for PRINT) will appear the word FULL, PART or STEP, indicating your program’s running speed. PART indicates a speed of 1, 2, 3 or 4.

Pun—RUNning At The Same Old Pace

There’s one final speed command in HOB, and it’s reserved for those special situations when you make a change in a program line

but want to RUN your program at a speed you've already set. Ordinarily entering RUN would reset PACE to full speed (5); and since you've changed something in the program, pressing [RETURN] won't allow program execution to continue.

PUN

fills the bill; think of it as meaning *keep PREVIOUS PACE and RUN*.

Back to Data

As we were saying before we started playing "Time Masters," a READ command will get a value for a variable from a DATA list. The same laws that apply to assigning values to variables by a LET statement apply to READ/DATA. That means you can assign values to string variables via a READ statement. Add:

25 DATA STUDENT

65 READ B\$

115 PRINT "THE ONLY STRING IS";B\$

Set PACE to STEP (or be ready to press the [SPACE] bar) and PUN the program. While we want you to see that string variables and READ/DATA can go together, we want you to see things happen via another special HOB screen; one that lets you see how DATA items are used up.

The Data Screen

Press the [T] key to get you to the CTRACE SCREEN. Press the [SPACE] bar a few times until you get to the first READ command at line 40. Then press [D]; you'll find yourself at the DATA screen (Press [Q] and note the D at the upper right). Here you'll find all the DATA lists (see figure 5-B) in the program.

The item in inverse video is the next one to be taken by a READ statement. Press the [SPACE] bar a few times; each time that a new item appears in inverse video, you can be sure that the item before it has just been READ. Flip back and forth between this screen and CTRACE to see the command lines as they occur. Speed things up by pressing the [→] key a couple of times.

Avoid Variables "Mismatch"

To avoid introducing pesky bugs into your program you must be careful not to assign a string to a numeric variable. Try this:

65 READ B

RUN

After trying its best to RUN your program, HOB will respond with:

B="STUDENT"

READ-DATA MISMATCH

Look at the upper left of the screen; HOB tells you where to look in your program for the error it found. When *READ* looks for a value, be it string or numeric, it looks at *DATA* statements in order of their line numbers. The first *DATA* statement is at line 20. Each time the computer gets a *READ* command it goes and takes the next available piece of *DATA* and marks it as being used up. There is enough *DATA* in line 20 for 9 *READ* commands (in this case, one *READ* command executed 9 times via a *FORLOOP*). The 10th *READ* command comes in line 65; a numeric variable in search of a value. But the next available piece of *DATA* is in line 25; and it's a group of letters. Letters don't go into numeric variables. Since *B* is a numeric variable the program aborts (or crashes, as we computer fanatics say). Set *PACE* to 1 (a speed of one command execution per second or so) and *PUN* the program again; watch it on the *DATA* screen until it crashes. Now *LIST* the program (see figure 18).

Rather than fixing this bug (accomplished by changing line 65 back to *READ B\$*), just eliminate line 25. Then *PUN* the program again and meet another *READ/ DATA* bug.

Restore

Applesoft's equivalent error message will be *OUT OF DATA*.

READ can't *READ DATA* that isn't there so you get a *READ IS OUT OF DATA* error in line 65; that's where the computer tried to pick up a value for *B*. There are at least two ways to solve this problem: add more *DATA* or make the computer reread old *DATA*. You add more *DATA* by typing a new *DATA* line:

25 DATA 87

But don't do it. Instead make the computer use *DATA* that's already been used by invoking a new command:

62 RESTORE

Watch the program *RUN* under the *DATA* screen; the *DATA* pointer will move back to the first *DATA* item when the *RESTORE* command is executed.

After you *PUN* this program, predict what the value of *B* will be. Then test your prediction either by looking at *SYMBOLS* or by typing *B* and pressing [RETURN]. Then come back here.

B should hold 10, the first value in the *DATA* line. Since all the *DATA* has been *RESTORED* to a pristine state, the computer treats this *DATA* as fresh and new and looks at it with new eyes. As far as the machine is concerned, this *DATA* is being *READ* for the first time.

```

65      31 FULL

65 READ B

B=STUDENT
READ-DATA MISMATCH
>LIST

10 LET S=0
20 DATA 10,15,36,123,47,12,15,13,19
25 DATA STUDENT
30 FOR X=1 TO 9
40 READ A
50 LET S=S+A
60 NEXT X
70 PRINT B
80 PRINT "THE NUMBERS' SUM IS";S
90 PRINT "THE ONLY STRING IS ";B$
100 FOR X=1 TO 100
110 NEXT X
120 END

```

FIGURE 17

```

10      0 STEP

20 DATA 10,15,36,123,47,12,15,13,19
25 DATA STUDENT

```

FIGURE 18

ROLLOUT this program; we call ours **FORLOP**. Then experiment with **READ/DATA** for a while. Don't forget to do the exercises at the end of the chapter.

Rules For READ/DATA

1. DATA lists may appear anywhere in a program. They may be separated from each other by other kinds of program lines or they may follow each other sequentially.
2. Items in a DATA list must be separated by a comma.
3. Strings and numbers may appear in the same DATA statement.
4. Strings in DATA statements need not be enclosed in quotes. Quotes **MUST** be used if commas are to be included in the DATA string.

5. READ statements must follow the general rules of syntax (that is, STRINGS may not be assigned to NUMERIC variables, etc.).
6. For every READ command there must be a matching DATA statement of corresponding type. But you don't have to use up all available DATA items.
7. Several variables may be READ from the same command line (the form is: READ A,C\$,J9, etc.)
8. Used DATA lists may be reused through the RESTORE command. RESTORE reuses all DATA from the earliest numbered DATA line and cannot be selectively set at the start of a particular line.
9. Up to 22 DATA lists (program lines beginning with the command DATA) are allowed per program.

There is no explicit limit to the number of DATA lists that appear in an Applesoft program.

Chapter Review

Keys:	[SPACE] bar	[←]	[→]	[t]	[x]
	[,]	[D]	[RETURN]	[+]	[−]
Commands:	READ	DATA	RESTORE	PACE	
	PUN				
Screens:	CHRONOLOGICAL	TRACE	DATA		
Terms:	TIME	"0"	SPEED	SINGLE STEP	
	MISMATCH	CRASH	DATUM		
	DATA LIST	CTRACE	STEP	FULL	
	PART				

Practice Exercises

1. Find and fix the bug in the following program:
10 DATA 1,4,5,SAM,MARTHA,12
20 READ A,B,C,D\$,E,F
30 PRINT A;B;D\$
9999 END

2. Name three ways of slowing down the following program so that a number is PRINTed approximately once every 3-4 seconds:

```
10 FOR N1=1 TO 5
20 READ N
30 PRINT N
40 NEXT N1
50 DATA 5,12,17,12.5,956
60 END
```

3. How could you assign the following sentence to a SINGLE string variable without using LET? Write a three-line program that does it:
THIS IS IT, NO?

4. Write a program using READ/DATA that assigns the word "REPETITIOUS" to 5 different string variables. "REPETITIOUS" may appear only ONCE in the program. Check your results by looking at the SYMBOLS table. Single step through your program watching the variable assignments through the CTRACE screen.

5. Using READ/DATA and a FORLOOP or a DOLOOP, write a program that will list the letters of the alphabet with their corresponding numeric positions; it should look like this:

A	1
B	2
•	•
•	•
Z	26

CHAPTER 6: INPUTS AND INTERACTIVE PROGRAMMING

So far we've seen two ways of getting values assigned to variables; **LET** and **READ ... DATA**. There's a third way that's really different from these two; the **INPUT** statement. What makes **INPUT** unique is that it gets values for variables from the person operating the computer rather than from within the program itself.

The Input Statement

```
NEW
10 INPUT A
20 INPUT A$
999 END
RUN
```

When you **RUN** this program, the **PRINT** screen appears and you are presented with a question mark. The question mark is a prompt telling you the computer wants something; it was produced by the **INPUT** statement in line 10. In this case, the computer wants you to type a number. Give it a [4] and press [RETURN]. Now there's another question mark on the screen, this one produced by line 20's **INPUT** statement. This time the computer wants a string. Type in **FRED** (actually you could type in anything, since strings can be any set of characters) and press [RETURN].

Now that you're back at the **COMMAND** screen, summon the **SYMBOLS** table. You'll find **A** and **A\$**, with 4 and **FRED** properly assigned.

INPUT follows the syntax rules of all variables: string **INPUT**s want alphabetic characters, numerics or special characters; numeric **INPUT**s want numbers only. As usual, you can identify string **INPUT**s by the dollar sign (\$) attached to the end of the variable name. **RUN** the program again and try to type alphabetic characters in response to the first question mark prompt. You'll see that **HOB** won't let you do it.

Interrupting Execution During Inputs

You may interrupt program execution while the computer is waiting for a response to an **INPUT** command by pressing [ESC]. The one

restriction is that the cursor must be at the first INPUT position. For example, if you have begun to type an answer to an INPUT question, you must either backspace using the [←] or enter a [CTRL][X] (see Chapter 1); then you can press [ESC] and program execution will halt. If the INPUT prompt has just appeared, you may immediately press [ESC] to stop the program.

Humanizing Your Programs

One of the problems with the little program we just used is that if you hadn't typed the program yourself, you wouldn't know what the question marks meant or what the computer wanted. It would be better to use PROMPT LINES to help out whoever was operating the computer:

```
5 PRINT "PLEASE TYPE A NUMBER"
15 PRINT "GREAT! HOW ABOUT A NAME?"
RUN
```

Now it's easy to tell what the computer wants. Anybody can work with *this* program, even people who don't know anything about programming.

We can go a step further in making this little program more understandable to people by getting rid of that question mark in the far left column. **EDIT** line 5 and add a space before the endquote, and a semicolon at the end. It should look like this:

```
5 PRINT " PLEASE TYPE A NUMBER";
```

The semicolon does two things here; it makes the *INPUT* response appear on the same line as the prompt, and it prevents the appearance of the question mark ordinarily displayed by the *INPUT* command. The extra space before the endquote is for clarity; if we didn't have it, the number typed by the user would be squashed against the end of the word **NUMBER**.

You can do the same thing with line 15 if you want to; but because of line length restrictions in HOB, you'll have to break this prompt into two separate lines. Experiment with this one on your own.

Chances are many programs that you write will be interactive. That is, they will need to get information from the person operating the program. Also, chances are that most people who use your programs won't be computer programmers. It's up to you to make sure that people unfamiliar with computers will be able to use your programs comfortably; rather than computerizing the computer operator, your job will be to humanize your programs. We'll show you

Applesoft allows string prompts to be contained *within* input statements; (e.g. INPUT "PLEASE TYPE A NUMBER";N).

how to do that throughout the rest of this manual. In fact, every program from now on will be humanized and most of them will be interactive.

Constructing An Interactive Program

Let's construct an interactive program that computes and reports the sales tax on a given item. Computing sales tax is a pretty simple procedure; take the cost of the item and multiply it by the tax rate. If a book costs \$5.00 and the tax rate is 5%, then the tax on the book is \$5.00 times .05, or 25 cents. The total cost of the purchase is the price of the item plus the tax; in this case \$5.00 plus 25 cents for a total of \$5.25.

We'll use the numeric variable **T** to stand for the *tax rate* (which we assume to be 5%). If we say that the variable **P** will stand for the *price* and that **T1** will be the *computed tax*, then **T1 = P * T** (*COMPUTED TAX EQUALS THE PRICE TIMES THE TAX RATE*). Variable **C**, which will stand for *total cost* will be **T1 + P** (*TOTAL COST EQUALS THE COMPUTED TAX PLUS THE PRICE*). That's all there is to it. Put this into program lines and you can compute taxes all day:

```
10 LET T = .05
20 INPUT P
30 LET T1 = P * T
40 LET C = T1 + P
50 PRINT T1
60 PRINT C
999 END
```

RUN

Use any number you like in response to the ? prompt. HOB goes back to the COMMAND screen when it finishes running a program. In order to see what the program produced on the PRINT screen, enter:

[CTRL] [P]

The first number displayed is the one you typed in. The second number is the tax, and the third is the total cost (tax plus price).

Now let's take this crude but effective program and make it more useful by giving it *PROMPT LINES*, thus making it more fit for human consumption. Don't type in **NEW**; we'll build around what we already have (lines that you need not type again will appear in parentheses). As we go along we'll be introducing several new commands and techniques; we'll explain each technique as we come to it:

```
(10 LET T=.05)
12 PRINT TAB(12);"TAX FIGURE-OUTER"
14 PRINT
```

Tabbing

TAB(n) moves the cursor to the nth position in the line. It always comes after a PRINT statement and is immediately followed by a semicolon and either a string or a number; the string or number following it will be displayed on the PRINT screen. The variable n is any whole number between 1 and 255. We use it here to center the words TAX FIGURE-OUTER on the top of the PRINT screen.

The PRINT statement in line 14 has no string or number to PRINT after it; therefore it will PRINT a blank line. We use it here to make the screen display more appealing to the eye (humanized programmers are big on aesthetics).

```
16 PRINT "ITEM'S COST, PLEASE";
18 PRINT "(JUST THE NUMBERS):"
```

These two lines tell the computer operator what to type. Line 18 is a warning not to type the dollar sign.

```
19 PRINT
(20 INPUT P)
25 PRINT
(30 LET T1=P*T)
```

These two PRINT statements display empty lines around the PRICE typed in response to the prompts in lines 16 and 18. Again, their use is aesthetic.

```
35 PRINT "THE TAX IS $";T1
(40 LET C=T1+P)
45 PRINT "THE TOTAL IS $";C
50 PRINT
60 PRINT "ANOTHER TAX?"
70 INPUT Q$
80 PRINT
90 IF Q$="YES" THEN 14
```

Lines 35 through 45 display the results of the computations. Line 60 wants to know if the computer operator has another tax that needs figuring. If the answer is YES then line 90 sends the computer back to line 14 to start the process again. If the answer is anything else, the computer looks for the next program line:

```
100 PRINT "IT'S BEEN GRAND";
110 PRINT "COMPUTING FOR YOU!"
```

Applesoft allows you to set the TAB to (0), which will act as a TAB (256).

WARNING: Setting tab to (0) will cause your program to end suddenly. This eventuality can be most embarrassing and should be avoided.

```
997 FOR Z=1 TO 200
998 NEXT Z
(999 END)
```

Lines 100 and 110 are the “goodbye” lines; they bring the program to some kind of a logical close and are there to make the program more friendly (which computers really ought to be).

Controlled Listings

When listing in Applesoft, [CTRL] [S] interrupts the listing, while [CTRL] [C] aborts it.

LIST the program now to see what it all looks like. If it goes by too quickly for you (there are more lines to the program than will fit on the screen) press the [SPACE] bar. Pressing the [SPACE] bar during a listing stops the listing temporarily; pressing it thereafter single steps through the listing. Pressing [RETURN] while a listing has been interrupted by the [SPACE] bar will cause the listing to continue at full speed, and [ESC] will cancel the listing entirely. If that’s confusing, try it and see what happens. In fact, you’d better try it even if everything is clear; we’ve been known to prevaricate.

Renumber

Applesoft requires special programs, called utilities, to renumber. There is a **RENUMBER** program on your DOS 3.3 master diskette.

When you get into larger programs with lots of additions, the gap between lines starts to shrink. If you wanted to insert four or five lines of code between lines 16 and 18, for instance, you’d be in a world of trouble. Luckily HOB has a built-in feature called **RENUMBER** to space the lines out. To make **RENUMBER** work, use the formula

RENUMBER base, increment

where **base** represents what the first number in the program will be (that is, the *lowest line number*) and **increment** is the *gap between lines*. Renumber the present program starting with line number 300 with gaps of 15. Enter:

```
RENUMBER 300,15
```

All the *referenced lines* are also renumbered. A referenced line is a *line referred to by the current line*; if line 100 in a program says **GOTO 150** then 150 is the *referenced line*. The former line 90 in this program (now line 540) used to refer to line 14; now it refers to line 330, the old line 14’s new number.

If you had typed the command **RENUMBER** and pressed [RETURN] without entering in any parameters (that is, without saying what the **base** or **increment** should be), the lowest line number in the program would have been 10 and the lines would have gaps of 10 between them. Try it; then save the program to disk (see figure 19.).



So far, you've learned three ways to assign values to variables: **LET**, **READ/DATA**, and **INPUT**. If you have a set of game paddles plugged into your computer, you have yet another way.

Turn the knobs on the game paddles to about the middle of their settings. If you have game paddles that have levers instead of knobs then set the levers to the center position. Then enter this little program:

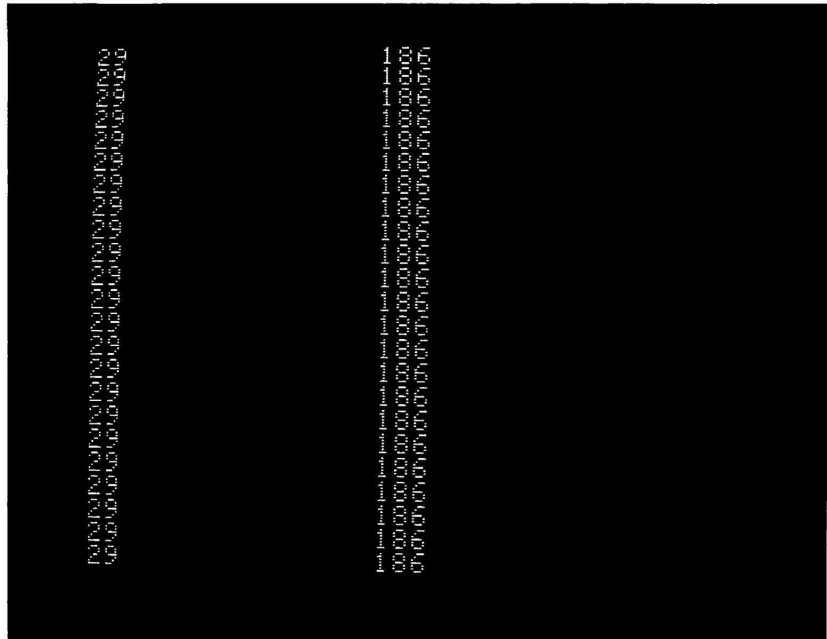
RUN

Now take either paddle and slowly turn it just a bit to the left and then to the right and keep your eye on the screen. The final number in either the left or the right column will start to change. If the left-hand column changes, you are holding $PDL(0)$; if the right-hand column changes, you've got $PDL(1)$. The idea is to locate $PDL(1)$. When you've done that, it's not a bad idea to mark the paddle in some way (a piece of masking tape on top of the knob will do) so you can distinguish it from $PDL(0)$.

Now that you've got PDL(1), turn the knob all the way to the left; the number 0 should appear. Turn it to the right *slowly* and the numbers should increase up to 255. The numbers in the left column won't change; they're controlled by the other paddle.

If the numbers are going by too quickly for you, remember that you can slow down the speed of the program's execution by pressing the [←] key.

FIGURE 20



Now take the other paddle and turn it all the way to the left. The program should stop (if it doesn't you may have entered the program incorrectly; press the [ESC] key).

PDL(0) and PDL(1) are the names the computer recognizes for the paddles (being strange sorts, computers count up from 0 rather than from 1 like the rest of us). As you've already seen, depending on the position of a paddle's knob a paddle will hold a value somewhere between 0 and 255. In that sense you can treat the terms PDL(0) and PDL(1) as special kinds of variable names that get their value from the position of the paddle knobs.

With that in mind, take a look at the program listing. Line 10 displays the value of the variables PDL(0) and PDL(1). Whenever you change the knob position on either paddle the number displayed

will change. It's as if there were a pair of invisible lines somewhere in the program that said

```
INPUT PDL(0)
INPUT PDL(1)
```

Line 20 is the way out of the program. When you turn PDL(0)'s knob all the way to the left, its value is zero and the program goes to line 999, the END line. We chose PDL(0) and the value zero arbitrarily; you could use PDL(1) and whatever value you wanted to accomplish the same thing. Line 30 sends the computer back to start the loop over again. Line 999 (as usual) ends the program.

You can assign the values of the paddles to other variables or treat them like you would any variable or math function. Put these two lines into your program:

```
15 LET A=PDL(0)
16 LET B=PDL(1)
```

Be sure to change the knob position on PDL(0) so it's not set all the way to the left (otherwise the program will stop immediately). Then **RUN** the program and press [T] for the CTRACE screen. Watch the values of **A** and **B** change as you turn the paddle knobs.

Program Without Comment

All we'll tell you about this program is: you work it with the paddles and you can end the program without pressing [ESC]. Good luck!

```
10 LET A$="BLEEP"
20 LET B$="BLOOP"
30 LET A=PDL(0)
35 IF A=0 THEN A=1
40 LET B=PDL(1)
45 IF B=0 THEN B=1
50 PRINT TAB(A);A$
60 IF A=1 THEN 90
70 PRINT TAB(B);B$
80 GOTO 30
90 IF B=1 THEN 999
100 GOTO 30
999 END
```


Chapter Review

Keys:	[SPACE] bar	[ESC]	[RETURN]
Commands:	INPUT	TAB(N)	PRINT RENUMBER PDL(0) PDL(1)
Terms:	HUMANIZE	INTERACTIVE PROMPT LINES	AESTHETICS REFERENCED LINES PADDLES CONTROLLED LISTINGS
Shorthand:	<i>base</i>	<i>increment</i>	

Practice Exercises

1. Write a program that will:
 - a. find out the first name of the computer operator
 - b. tell the operator something complimentary, addressing him or her by name
2. Write a program to accept values for three numeric variables from one of the game paddles. Part of the problem is to insure enough time to change the setting on the game paddle so that different values can be sent to the computer. You might want to add warnings to yourself when it's time to change the paddle setting.
3. There are AT LEAST four errors in the following program, including sloppy screen display stuff. Find the errors WITHOUT entering the program. Then enter and RUN it with the errors corrected:

```
10 PRINT "HI! WHAT'S YOUR NAME?"
20 INPUT A
30 PRINT TAB(PDL(5));"WELL,";A$;
40 PRINT "WELCOME TO COMPUTING!"
50 END
60 FOR Z=1 TO 10
70 NEXT Z
```

4. What are the differences among a TARGET LINE, a REFERENCED LINE and PROMPT LINE (warning: trick question)?
5. Write a humanized interactive program that lets the operator play with the game paddles. The program must:
 - a. include instructions for the operator
 - b. multiply the setting on PDL(0) by the setting on PDL(1)
 - c. display the numbers being multiplied and the results
 - d. do (b.) so that the operator knows what the displayed numbers mean
 - e. continue until getting some "DONE" signal

CHAPTER 7: ARRAYS

In the second chapter we said that we'd be talking about a special kind of variable that acted differently from regular numeric and string variables. That special kind of variable is the *ARRAY* or *SUBSCRIPTED VARIABLE*. In HOB, arrays are like numeric variables except for one special difference: a single variable name can refer to a whole list of variables.

What Arrays Are Made of

Array variable names consist of two parts; the *ARRAY TITLE* and the *ELEMENT* (also called the *SUBSCRIPT*). You'd refer to the fifth position of an array called *Z* as *Z(5)*. The *array title* is *Z* and the *element* is (5). If there were a variable called *N* that had been defined as holding the value 5 we could refer to the same element by talking about *Z(N)*. The *array title* part can be any single letter of the alphabet; the *element* part is either an integer (a whole number, no decimals or fractions) or a numeric variable. The element always is written between two parentheses.

Enter this code in IMMEDIATE EXECUTION mode:

```
LET A=2
LET B(1)=12
LET B(A)=5
LET B(3)=B(1)+B(2)
```

Check what each array element holds. The computer sees each array element as a separate variable, so you'd ask the computer to show you the value for each element in the same way you'd ask it to show you what a variable holds: enter the variable and press [RETURN] (a good way to test your understanding is to predict what the results will be before you ask the machine to tell you). Enter

B(1)

And you'll see

THE RESULT IS 12

Now try

B(2)

And you'll get (hopefully)

THE RESULT IS 5

You said to **LET B(A)=5**. Since $A=2$, then to say **LET B(A)=5** is the same as saying **LET B(2)=5**. Now enter:

B(3)
THE RESULT IS 17

Array Charts

If you were to chart out this array on paper it would look like this:

ARRAY B

1	12
2	5
3	17

Don't get confused between the *name* of the array element and the *value* the array element holds. Array **B** element #3 holds the sum of the values of array **B** element #1 plus array **B** element #2.

Because a variable can be substituted for a number within an array subscript, we can do this:

```
10 FOR Q=1 TO 7
20 READ D(Q)
30 NEXT Q
40 DATA 6,12,18,24,30,36,42
999 END
```

Here's a chart of the array with the values filled in:

ARRAY D

1	6
2	12
3	18
4	24
5	30
6	36
7	42

As the program runs, it assigns values to seven elements (numbers 1 through 7) of array **D**. Which element is currently being assigned a value is dependent upon the value of **Q**; because **Q** is the **index** of a **FOR** LOOP whose **lowlimit** is 1 and whose **highlimit** is 7, the value of **Q** will range from 1 to 7. The first time through the loop, **Q**

will hold the value 1; that means that the first item in the *DATA* list *READ* by line 20 will be assigned to the array variable D(1). Since the first *DATA* item is 6, D(1) will be assigned the value 6. The second time through the loop, Q will hold 2, so the second *DATA* item (12) will be assigned to D(2). And so it goes to the end of the *FORLOOP*.

Add this line to your program and see what happens when you run it:

25 PRINT D(Q)

As the values are assigned to the different array elements by line 20, line 25 displays the values for you.

RUN the program again. This time call the *CTRACE* screen (press [T]) so you can watch the computer assign values to the array variables (see figure 21). Single-step your way through the program (use the [SPACE] bar) to watch things happen. Then will all things become clear and all mysteries be revealed.

FIGURE 21

```

          999      16 FULL
D(4)=24
  30  NEXT Q
Q=5

  20  READ D(Q)
D(5)=30
  30  NEXT Q
Q=6

  20  READ D(Q)
D(6)=36
  30  NEXT Q
Q=7

  20  READ D(Q)
D(7)=42
  30  NEXT Q
Q=8

  40  DATA 6,12,18,24,30,36,42
999  END

```

Dimensioning Arrays

Most BASICs (HOB included) automatically allow you 11 elements per array; in the above array you can have values assigned to elements D(0) through D(10) (in case you forgot, computers consider 0

to be the first number. Each array starts with an element #0. If we wanted an array to have more than that many elements (say, 51 instead of 11) we would have to tell the computer:

5 DIM D(50)

Just as in a *FORLOOP* the computer assumes that the STEP is +1 unless you tell it otherwise; in arrays the computer assumes a DIM of 10. If you use a DIM statement it must appear in the program before the first time the array being DIM'ed is used.

Two-Dimensional Arrays

Our original D array is formally called a one-dimensional, 11 element numeric array. HOB also allows two-dimensional arrays. Assume that the DIM statement said

DIM J(4,3)

A chart of the J array might look like this:

ARRAY J

	0	1	2	3
0				
1				
2				
3				
4				

This is a two-dimensional, twenty element array. It's two dimensional because we have elements that need to be *addressed* (specified) by giving two numbers instead of just one; the array goes in two directions (across and down). It has twenty elements because there are five down and four across (5 times 4 is 20). This includes the elements in the row and the column marked 0. We can assign values to the array just like we did with single dimensional arrays, except that here we have to specify the element by giving two numbers.

To explain further, assume you wanted to assign the value 35 to

Applesoft allows arrays up to 88 dimensions.

element 3,2 and you wanted element 4,3 to hold 57. You would say

LET J(3,2)=35

LET J(4,3)=57

The array would look like this:

ARRAY J				
	0	1	2	3
0				
1				
2				
3			35	
4				57

You might think of J(3,2) as row 3, column 2 and J(4,3) as row 4, column 3 to make it easier for you to visualize how variables are assigned to arrays. Only those two positions have been filled; the rest are still empty.

Loading 2-D Arrays with Read . . . Data

Time for the heavy stuff: here's a program that uses *nested FOR-LOOPS* and some *DATA STATEMENTS* to *LOAD* (assign values to the elements of) a two-dimensional array. **ENTER** and **RUN** it.

```
10 FOR A=1 TO 4
20 FOR B=1 TO 3
30 READ J(A,B)
40 NEXT B
50 NEXT A
100 DATA 11,22,33,44,55,66,77,88,99
110 DATA 111,222,333
999 END
```

Here's what the value assignments would look like:

	ARRAY J			
	0	1	2	3
0				
1		11	22	33
2		44	55	66
3		77	88	99
4		111	222	333

Running the program and watching things happen with CTRACE, we actually see J(1,1) being assigned 11, J(1,2) getting 22 and so on. If the nesting of the *FORLOOPS* has you confused use the [F] key to switch to the *FORLOOP* screen; note which *index* goes with which loop. Here's a diagram of what happens as both the outer loop (A) and the inner loop (B) increment:

		these values
	A B	READ
	A=1	↓ ↓ ↓
	B=1	J(1,1)=11
	B=2	J(1,2)=22
nested	B=3	J(1,3)=33
B		
loop	A=2	
	B=1	J(2,1)=44
	B=2	J(2,2)=55
	B=3	J(2,3)=66

Filling Arrays Arithmetically

Here's a variation on the last program. It uses nested *FORLOOPS* again but rather than getting its values from *DATA* lists it *LOADS* the array using arithmetic based on the values of the *FORLOOP* indexes.

First we'll show you the program to type in and run. Then we'll give you an array chart filled out with all the values. You'll have to trace through the program yourself to see how the values were assigned. Don't forget to use the different PROGRAM TRACKING SCREENs to help you:

```

10 FOR A=1 TO 4
20 FOR B=1 TO 3
30 LET J(A,B)=A*B
40 NEXT B
50 NEXT A
999 END

```

	ARRAY J			
	0	1	2	3
0				
1		1	2	3
2		2	4	6
3		3	6	9
4		4	8	12

The OPTION BASE Command And Those Pesky Zeros

Row 1 and column 1 both begin with 0; as we said before, computers begin counting from 0 rather than 1. If this is confusing to you then ignore it. Unless you actually use the 0 element (or in this case elements) in an array, its presence won't affect anything; you won't even know it's there. In fact, HOB gives you a command to use that will make the 0th elements *not* be there. The command is

OPTION BASE 1

and it must appear as the first line in a program. If you later decide you want to use the 0th array element you must remove the com-

There is no OPTION BASE command in Applesoft; all arrays begin with a 0th element.

mand from the program. Programs default to `OPTION BASE 0`, allowing a 0th element.

A Challenge: Using the 0th Elements

Add these two little lines to your program:

```
15 LET J(A,0)=0
35 LET J(A,0)=J(A,0)+J(A,B)
```

When you run the program you'll get four more elements of the array filled, `J(1,0)` through `J(4,0)`. The chart will look like this:

ARRAY J

	0	1	2	3
0				
1	6	1	2	3
2	12	2	4	6
3	18	3	6	9
4	24	4	8	12

This is kind of tricky stuff. Line 15 initializes elements `J(1,0)`, `J(2,0)`, `J(3,0)` and `J(4,0)` to zero as `A` changes in value from 1 through 4. Line 15 is the best place to do this initializing; `A` just got its value as the **index** of the `FORLOOP` and the `B FORLOOP` hasn't started yet. If we put the line inside the `B FORLOOP` then `J(A,0)` would be reset to zero in every single loop of the program.

Line 35 says

```
LET J(A,0)=J(A,0)+J(A,B)
```

This line comes at the bottom of the `B FORLOOP`; since `B` loops three times for every value of `A`, the value of `J(A,0)` increases three times. It's the final value we're after, the sum of `J(A,1)+J(A,2)+J(A,3)`. Eventually we get sums for all the elements in all the rows stored in the 0th position of the rows, called `J(A,0)`. Use HOB's various screens to watch it happen if you're at all confused (which is likely, though this is tough stuff).

Now, the real challenge is to write an addition to the program that will sum up all the columns and store the results in the three elements we'd refer to as J(0,B)! See the back of the manual for a solution (but only after you've tried for a while).

Rules For Arrays

1. All HOB arrays are numeric.
2. Unless stated in the first line of a program the OPTION BASE of arrays is assumed to be 0. The only other BASE is 1.
3. DIM statements (if used) must appear before the first reference to an array element.
4. One-dimensional arrays that have not been DIM'ed by a program command are assumed to be DIM'ed (10). UnDIM'ed 2-dimensional arrays are assumed to be DIM'ed (10,10).
5. Maximum DIM for a 1-dimensional array is (255). Maximum DIM for a 2-dimensional array is (255,255).
6. Arrays may not be reDIM'ed.
7. Array references are always made with whole numbers or integer variables (as opposed to fractions or decimal numbers). Fractional references are rounded and INT'ed by HOB; a reference to X(5.4) will look at X(5), while a reference to X(5.5) will look at X(6).
8. Array elements may be treated like any other variable in that they may be used for READ, INPUT, etc.

The maximum dimension size for an array in Applesoft is limited only by the amount of memory available.

Chapter Review

Commands:	DIM	OPTION BASE		
Terms:	ARRAY	SUBSCRIPTED VARIABLE		
	TITLE	ELEMENT	INTEGER	LOAD
	CHART			

Practice Exercises

1. What's the essential difference between an ARRAY and a "regular" numeric variable?
2. Using READ/DATA and a FORLOOP with an *index* variable P, load the following five numbers into an array called J: 53, 19, 60.5, 4, 17.
3. Using the array you just constructed, what is the product of J(3) times J(4)? How could you store that number in J(6)?
4. How many possible ELEMENTS are there in an array called Q, found in a program with the following program line:
1 DIM Q(5,9)
5. How would you eliminate the possibility of the above array having an element called Q(0,5)?

CHAPTER 8: OTHER WAYS OF BRANCHING

Here's a little program that demonstrates *CONDITIONAL BRANCHING* in a way we haven't seen before; we'll use it to introduce you to a new HOB screen, LIST TRACE. Type in the code and look at it for a while. See if you can figure out what it will do under what conditions before you RUN it:

```
5 PRINT "NUMBER, PLEASE";
10 PRINT "(1 TO 5):"
15 INPUT X
20 IF X=1 THEN 100
30 IF X=2 THEN 200
40 IF X=3 THEN 300
50 IF X=4 THEN 400
60 IF X=5 THEN 995
65 PRINT "YOU BLEW IT!"
70 PRINT A$
80 GOTO 5
100 LET A$="THIS IS LINE 100"
110 GOTO 70
200 LET A$="HERE'S LINE 200"
210 GOTO 70
300 LET A$="I'M LINE 300"
310 GOTO 70
400 LET A$="BIG LINE 400 HERE"
410 GOTO 70
995 PRINT
996 PRINT "CIAO!"
997 FOR L=1 TO 100
998 NEXT L
999 END
```

Now RUN it. When it asks for a number, enter 1, 2, 3 or 4. Play with it a bit; then enter a 5 to stop. Try entering an *out of range* number and see what line 65 is about!

"Playing" Computer

If you have trouble figuring out what this program is doing, pretend you are the computer. Read the program from the first line and

follow the instructions line by line. When you get to line 15 pretend that someone has entered a 1 and go through the program with X holding the value 1. Then do it again with X holding 2, and so on through 5. We personally guarantee that “playing computer” will help you understand how nearly any program works!

HOB's List Trace Screen

Set PACE to STEP and PUN the program. Then press the [L] key. You should see something like figure 22.

```

10      1 STEP
5 PRINT "NUMBER, PLEASE ";
15 PRINT "CL TO 5)";
INPUT X
IF X=1 THEN GOTO 100
IF X=2 THEN GOTO 200
IF X=3 THEN GOTO 300
IF X=4 THEN GOTO 400
IF X=5 THEN GOTO 500
PRINT "YOU BLEW IT!"
GOTO 10
PRINT "THIS IS LINE 100"
GOTO 200
PRINT "HERE'S LINE 200"
GOTO 300
PRINT "I'M LINE 300"
GOTO 400
PRINT "BIG LINE 400 HERE"
GOTO 500
PRINT "CIAO!"
L=1 TO 100

```

FIGURE 22

If you stopped where we did the computer was about to execute line 15. That's the line appearing in inverse video. Single-step through the program now by pressing the [SPACE] bar. Soon you're back at the PRINT screen; HOB goes there when it wants an INPUT. Give it a [1] and press [RETURN]. HOB will automatically return you to the LIST TRACE screen, where line 20 is now in inverse video. Care to guess what will happen when you press the [SPACE] bar again?

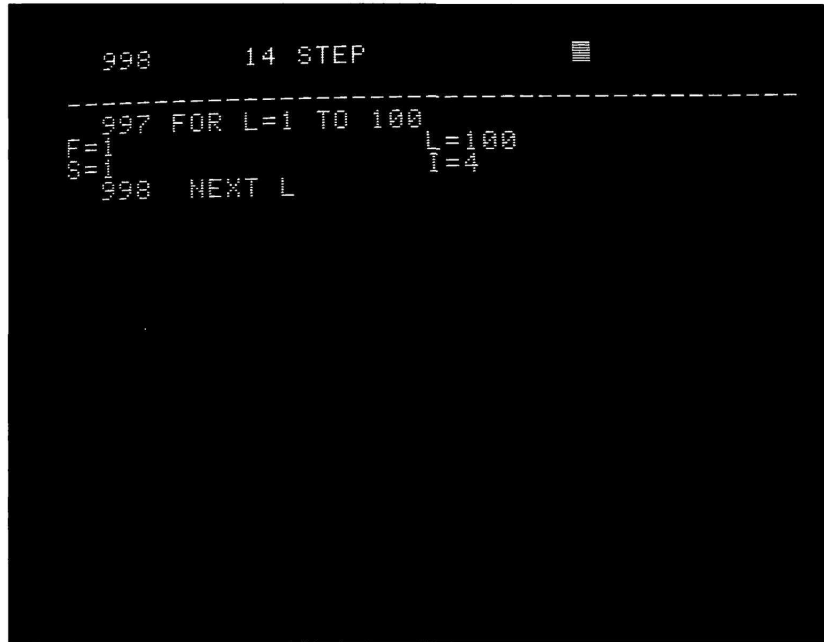
Single-step your way through a few more values for X before setting X to 5. Then something strange will seem to happen; LTRACE (for LIST TRACE) goes to line 996, 997 and 998 on successive presses of the [SPACE] bar. Then it seems to *hang* (computer term for not do

anything). Press the [SPACE] bar a few times to see what we mean. Something IS actually happening; we're in the midst of a *DELAY LOOP*.

Program Tracking Screen Review

Press [F] for the *FORLOOP* screen (figure 23).

FIGURE 23



```
998      14 STEP
-----
997 FOR L=1 TO 100      L=100
F=1                     L=4
S=1
998 NEXT L
```

Now press the [SPACE] bar a few more times and watch the **index** value change. Line 998, being the only line in the *L FORLOOP*, is repeating itself 100 times. You can't see any changes in the *LTRACE* screen because there are no changes! If you want to see changes press [T] to view the *CHRONOLOGICAL TRACE* screen (figure 8-C).

And take a look at the *PRINT* screen (press [P]) to see the "signoff" message (figure 25).

On var GOTO—Another Way of Branching

There's another way to do the same thing this program does using less code and going through fewer programming steps. Make the following changes:

```
20 ON X GOTO 100,200,300,400,995
DEL 30,60
```

```

998      18 STEP
998 NEXT L
L=2
998 NEXT L
L=3
998 NEXT L
L=4
998 NEXT L
L=5
998 NEXT L
L=6
998 NEXT L
L=7
998 NEXT L
L=8
998 NEXT L

```

FIGURE 24

```

NUMBER, PLEASE (1 TO 5):
73
I'M LINE 300
NUMBER, PLEASE (1 TO 5):
74
BIG LINE 400 HERE
NUMBER, PLEASE (1 TO 5):
72
HERE'S LINE 200
NUMBER, PLEASE (1 TO 5):
71
THIS IS LINE 100
NUMBER, PLEASE (1 TO 5):
75
CIAO!

```

FIGURE 25

Now run the program again to satisfy yourself that it does the same thing as the last version. Run it a second time using LTRACE (the [L] key) and single-step ([SPACE] bar) your way through it to see what it does.

In Applesoft, if the numeric variable exceeds the number of line numbers listed in the ON ... GOTO statement, that statement will be ignored.

Line 20 acts like a combination of the old lines 20,30,40,50 and 60. When you use **ON var GOTO** the value for **var** can be from 1 to the total number of line numbers following the **GOTO** statement. **Var** can be any numeric variable (including array variable) or expression. If the variable exceeds the number of line references an **OUT OF RANGE** error message will appear. The computer will branch to the line number **var** positions after the word **GOTO**. Each position is separated from its neighbors by a comma. In line 20 there are 5 such positions: 100, 200, 300, 400 and 995. **X** can therefore hold 1,2,3,4 or 5.

Avoiding “Bad” Inputs: Using Relational Operators

While we all would never type in a larger number than the instructions said we could, there are malcontents and revolutionaries out there who hate to follow directions. There are also people who press wrong keys by accident. We could wire the keyboard with 5000 volts of electricity to be delivered if too high a number were entered; that would certainly take care of the malcontents. Unfortunately that would also dispatch our Ancient Aunt Drusilla who sometimes has trouble reading numbers.

A compromise is possible thanks to the *RELATIONAL OPERATORS* to which you were introduced in Chapter 3. To remind you, the *RELATIONAL OPERATORS* are the symbols **<** and **>** and **=**, meaning **LESS THAN** and **GREATER THAN** and **HOLDS THE VALUE OF**. They compare the number or value of the variable to their immediate left with the number or value of the variable to their immediate right. Here's how you'd use them in the current program:

```
17 IF X<6 THEN 20
18 PRINT "SORRY— TOO HIGH!"
19 GOTO 15
```

Line 17 says *If the value for X is less than 6 branch to line 20*. It makes sure that the number INPUT isn't too high. Line 18, a message to the operator, only gets printed if **X** is 6 or higher. Then the computer branches back to 15 to give the operator another shot at it.

Similarly if we wanted to make sure that the number entered wasn't a zero, we could say:

```
17 IF X>0 THEN 20
18 PRINT "SORRY— TOO LOW!"
19 GOTO 15
```

The problem is how to protect against both kinds of errors. In the next paragraph we propose a solution. It's not the only solution, but it works. Before you read it, come up with your own idea and try it out. It might be better than ours (actually, anything that does the job is OK).

Here's one way: first, move line 20 to line 30. You can do that by rewriting line 20 again with 30 as its line number. Or, if you're as lazy as we are, do the following: **EDIT 20**. When the cursor appears at the end of the line, press and hold down the [←] key at the same time you press down and hold the [REPT] (for REPEAT) key; you'll find it just above the [←] key. The cursor will move all the way back to the beginning of the line. Replace the 2 with a 3. Then press and hold the [→] key at the same time you press and hold [REPT]. When the bells start ringing press [RETURN]. Sneaky, but effective.

NOTE: If you have an Apple IIe, just hold down [←]; repeating is automatic.

```
17 IF X<6 THEN 20
18 PRINT "SORRY—TOO HIGH!"
19 GOTO 15
20 IF X>0 THEN 30
24 PRINT "SORRY—TOO LOW!"
26 GOTO 15
```

Line 17 is the *maximum* test. If the number isn't too high, the computer goes to line 20 for the *minimum* test. If the number passes this test too then it must be a valid number and the computer can continue with the program. The malcontents have been subverted and Aunt Drusilla is protected from electrocution. You'd better run it and test our protection scheme, just to make sure. We'll keep the voltage meter handy.

The Ultimate Branch: GOSUB/RETURN

Sometimes there are long sections of code used repeatedly in a program. Typing the same program lines again and again is an inefficient use of your time and computer memory. Luckily there's a kind of COMMAND PAIR you can use to make things a lot easier on your fingers, the **GOSUB/RETURN** pair. To demonstrate it, we'll change a couple of the lines in the current program:

```
18 GOSUB 500
```

24 GOSUB 500

and add these lines:

```
500 PRINT "SORRY—ONLY NUMBERS";  
510 PRINT "BETWEEN 1 & 5 CAN"  
520 PRINT "BE USED. TYPE A 5 ";  
530 PRINT "TO END THE PROGRAM."  
540 PRINT  
550 RETURN
```

When you run this program, enter a 7 or some other number clearly out of range. This number will fail the $<$ test at line 17. Line 18 will make the computer *GO to a SUBroutine* at line 500 (the GOSUB command is an unconditional branch to the line number appearing after it). There the PRINT statements you just typed in at lines 500 through 530 will be displayed; at line 550 the machine will return from whence it came (line 18) and continue on its way. Note that you don't have to keep track of the line number that contained the GOSUB command; the computer remembers for you (by listing the number of the command line in a special index called the STACK) and finds its own way back.

You can begin to appreciate the GOSUB command when you consider a program you might write that would check for INPUT errors not just once but dozens of times. It's a lot better to type one line (GOSUB 500) over and over than it is to type five lines (lines 500-540) repeatedly.

Stacking Up GOSUBS

Applesoft allows GOSUBs to be nested up to 25 levels deep.

Like *FOR*LOOPS, *GOSUB*s can be nested. You can have GOSUBs within GOSUBs up to 16 levels deep. But make sure that every GOSUB has a RETURN and that every RETURN has a GOSUB.

Before we go on, **ROLLOUT** this program under the name **GOSUBSAMP**; we'll come back to it after we describe another PROGRAM TRACKING SCREEN, the GOSUB screen.

GOSUB Screen

The GOSUB screen, accessed by pressing [G] while a program is running or [CTRL][G] when the program has been interrupted, gives you information about the GOSUB stack (that is, list of the program lines containing active GOSUB commands) and displays the referencing program lines. To demonstrate this screen effectively, we'll need a program with lots of GOSUB commands. Hidden in the

depths of HOB is such a program. Summon it by entering the cross-hatch sign [#] followed by [G], thusly:

#G

Actually there are five such hidden programs, each designed to demonstrate the various PROGRAM TRACKING SCREENs. We'll use more of them in Chapter 11: GETTING THE BUGS OUT.

If HOB doesn't allow you to type more than the # symbol without beeping, the hidden programs somehow got clobbered. This can happen for a variety of reasons that we needn't be concerned about. Just restart HOB and try again. (See the introduction to this manual.)

This program does nothing but demonstrate itself; all the subroutines are only one line long! Don't look for high purpose here; there isn't any. Before running this program set PACE to 0; we'll want to single-step through it. This program has no PRINT statements in it; when you PUN it (which you should now do), you are faced with a blank PRINT screen.

Press [L]. The white line should be across line 10 (see figure 26). Now single-step only one command (press the [SPACE] bar once). The program advances to line 30, the location of the first subroutine. Now we're ready to look at the GOSUB screen. Press the [G] key. There's one line displayed here; the source line that sent the computer to line 30 of the program. Press the [SPACE] bar a few more times; watch the list of source lines grow. While you're at it, press the [Q] key to display the right side of the PROGRAM STATUS LINE. Figure 27 shows what your screen should look like.



FIGURE 26

FIGURE 27

The G on the far right side of the PROGRAM STATUS LINE indicates we're watching the GOSUB screen. To its left is another G followed by a number. This G means that there are active subroutines (that is, lines of code accessed by a GOSUB command), and the number tells how deep the subroutine stack is (how many subroutines have been accessed without encountering a RETURN command). Each time a RETURN command is executed, the G number will decrement (decrease by 1), and the command line listed at the bottom of the stack will disappear. Press the [→] four times to speed things up and to see the list shrink and grow. Go back and forth between the GOSUB and LIST TRACE screens to get a better sense of how subroutines work. Then press [ESC] to stop the program; it's written with an infinite loop built in! Go on to the next section to learn more about the GOSUB command.

GOSUB Errors

ROLLIN the GOSUBSAMP program. Now take out line 410. Then run the program and enter the number 4. The program will run for a bit and then crash with the messages 550 GOSUB UFLOW and RETURN & GOSUB STACK EMPTY. UFLOW is short for *underflow*.

We said earlier that the computer keeps track of where GOSUBs go so that it can get back when it encounters a RETURN command. To review a bit, when the computer encounters a GOSUB command it places the GOSUB's line number in a special index called a STACK. In the case of our program, it would put either 18 or 24 on the stack, depending on the line number that contained the GOSUB. When the computer comes across the RETURN command, it goes to the stack to see to where it should RETURN (it gets the ADDRESS of the line to which it must RETURN).

Because GOSUBs can be nested up to 16 levels deep there may be up to 16 numbers on the stack; the computer takes the last one put on the stack as the RETURN address. By the way, that's literal. If there were 15 RETURN addresses on the stack when the computer encountered the RETURN command, there would be 14 addresses left. It actually removes the address it's going to use. If there's no number there at all (that is, if the stack is empty) then an *underflow* condition exists and the computer doesn't know what to do. So the program stops and you get an error message.

The other possible error you might encounter is the **GOSUB STACK FULL** message. You'll get that one if you already have GOSUBs nested 16 deep and you attempt to GOSUB again. HOB can't handle more than 16 nested GOSUBs at a time. Remember that we're not talking about the number of GOSUB/RETURN sets you can have in a program; there's no limit to that. We're talking about the number of GOSUB commands the computer encounters before it comes across a RETURN command. Every time the computer comes across a RETURN command room is made for another GOSUB. In actual practice you will seldom come across this bug; few programs nest GOSUBs so deeply.

Run the program several times. First, do it straight without using any of the screens. Then use CTRACE, then LTRACE, in single step or at very slow speed. You can use PACE or [←] to set the speed. Watch the program on the GOSUB screen as well. The important thing is to see how the command works and how it affects the program flow. Use whatever method works for you. Try it with line 410 left out (so you get an error message) and with it back in (so you can see an intact program work).

After you've done that, you're on your own. Play around with the new things you learned in this chapter; then do the exercises.

Applesoft's error message for filling the GOSUB stack is **OUT OF MEMORY** error. This is usually caused by exiting a subroutine without using a RETURN statement.

Chapter Review

Keys:	[L]	[REPT]	[G]	[#]
Commands:	ON var GOTO	GOSUB/RETURN		
Terms:	PLAYING COMPUTER	INVERSE VIDEO		
	HANG	RELATIONAL OPERATORS		
	STACK OVERFLOW	UNDERFLOW		
	LTRACE			
Screens:	LIST TRACE	GOSUB		

Practice Exercises

1. What would be the result of entering the number 9 in response to the INPUT request in the following program segment?

```
10 PRINT "ENTER A NUMBER:"
20 INPUT A6
30 ON A6 GOTO 100,297,116,999
•
•
999 END
```

2. Predict what would happen if the following program were entered and RUN (but DON'T enter it; RUN the program in your head or on paper). What line is the source of the problem?

```
10 PRINT "THIS IS A TEST"
20 GOSUB 100
30 PRINT "THE TEST IS OVER"
40 GOTO 999
100 PRINT "HERE'S THE TESTED PART"
110 GOTO 20
120 RETURN
999 END
```

3. Find the fatal "error of omission" in this program. Then add one line of code to correct the error. AFTER you've found and fixed the bug, enter the program and test it:

```
10 PRINT "GOING TO A SUBROUTINE ..."  
20 GOSUB 100  
30 PRINT "... NOW I'M BACK!"  
100 REM HERE'S THE SUBROUTINE  
110 PRINT "... AT THE SUBROUTINE ..."  
120 RETURN  
999 END
```

4. Why are the operators "<", "=", and ">" called RELATIONAL operators?
5. Write a program that will have the operator guess the age of the programmer within 5 tries. The program must contain the following:
1. variable A holding the programmer's age
 2. variable I used to hold the guess INPUT by the operator
 3. variable C holding the COUNT of guesses made
 4. subroutines with messages to the operator saying if the guess made was too low or too high
 5. a way of limiting the number of guesses to 5
 6. various congratulatory messages to winners depending on how few guesses it took
 7. a message notifying a loser that all the guesses have been used up

CHAPTER 9: GETTING FUNCTIONAL

In this chapter we'll investigate a few of the special functions we didn't cover in Chapter 1. Specifically we'll look at **RND** (which generates a pseudorandom number) and **RANDOMIZE** (which makes the random number generated less pseudo), **INT(n)** (which returns the integer value of a number), **BTN(n)** (which tests if one of the paddle buttons has been pushed) and **DEF FN** (which allows you to define your own numeric functions). We'll also describe some ways to add sound to your programs, and a special way to renumber them.

Randomness (sort of)

Make sure that your computer's memory is clear of any leftover stuff from the last program by entering **NEW**. Then enter the following command IN IMMEDIATE MODE:

RND

Your machine will respond with some less-than-one decimal number. Enter **RND** again and you'll get another number. Keep entering **RND** and you'll continue to get numbers less than one, all of them different. **RND** is short for *RANDOM*; the computer picks a less-than-one number seemingly at random each time the command is invoked. We say seemingly because there's actually an internal order to the numbers generated. To make this clear, note the numbers generated by **RND** that are now on your screen. Now enter **NEW**. Now produce some random numbers again using **RND** a few times.

You'll see that the numbers produced are the same ones, in the same order, that were produced by the first series of **RND** commands. Try it again: enter **NEW** and then a series of **RND** commands and the same series will repeat. The numbers are seemingly random; that's why it's called a PSEUDORANDOM NUMBER GENERATOR. While getting the same random numbers all the time has some application to the sciences, it isn't very useful for most situations. Take games, for instance; can you imagine a game of BACKGAMMON or CRAPS where you always get the same series of numbers every time you throw the dice? In a few paragraphs we'll introduce you to a DEFERRED MODE command that, when used in conjunc-

In Applesoft, the function which produces a new random number is **RND(1)**.

tion with **RND**, overcomes this restriction and produces more realistically randomized numbers.

So that we can demonstrate some of the ways **RND** can be used we'll need to "capture" a number it produces. We can do that by assigning a value of **RND** to a variable. Again in immediate mode enter

LET A=RND

A

your computer will respond with

THE RESULT IS .20799392

and if you enter

A*10

your computer will display

THE RESULT IS 2.0799392

NOTE: Your result might be different.

The Integer Function

Now try this new command:

INT(A*10)

THE RESULT IS 2

INT(n) is a function that returns the integer part of (n). That is, it returns the whole number part of (n) with any fraction or decimal part dropped. Used in conjunction with **RND** you can do some really interesting things, like developing a computerized game of CRAPS, for instance.

Enter this:

INT(A*6)

HOB will answer

THE RESULT IS 1

If you can't figure out how HOB got that answer, turn on **FINETRACE** and enter **INT(A*6)** again.

Using **FINETRACE** to watch the process, we got

FINETRACE

INT(A*6)

INT(.20799392*6)

INT(1.2479635)

1

THE RESULT IS 1

By the way, if you summon the right side of the **PROGRAM STATUS LINE** by entering a **[CTRL][Q]** you'll see an **F** displayed about eight spaces from the right. This letter indicates that **FINETRACE** has

been turned on. The F will disappear when you turn FINETRACE off. Turn off FINETRACE (NOFINETRACE) and try a few rounds of

INT(6*RND)

Eventually you'll notice that you get the range of whole numbers between 0 and 5 inclusive. You'll never get a number higher than 5; the highest number that RND can produce is .999 . . . , and $6 * .999$. . . is 5.99999999994 (or so), and the integer of that is 5. If we change the expression slightly to

INT(6*RND)+1

the range of numbers changes to 1 through 6, the same numbers that appear on dice. *Such a coincidence!*

Dice Game Simulation

We'll use the computer to do a series of computations for us. Enter the following program (notice the new command in line 10):

```

10 REM DICE THROW SIMULATION
20 PRINT "HOW MANY ROUNDS?"
30 INPUT N
40 FOR X=1 TO N
50 PRINT "YOU ROLL A ";
60 PRINT INT(6*RND)+1
70 NEXT X
995 FOR Z=1 TO 300
996 NEXT Z
999 END

```

REM

The command **REM** in line 10 is short for *REMARK*. It tells the computer *ignore the rest of this line, it's only for humans*. REM lines are used by programmers to write notes to themselves about how their programs are constructed. As you have no doubt concluded by now, programming can get complicated. The best programmer in the world will have trouble understanding his or her code six months after the program has been written, especially if the program is a long one. REM lines allow the programmer to make internal comments throughout the program. There is no limit to the number of REM lines a program can have. Just remember that when the computer sees a line beginning with the command REM, it ignores it.

Now RUN the program.

After you've played with it a bit, run the program again and give it a

5 when it asks how many rolls you want. Notice the numbers it gives you. Now run the program again and give it whatever number you want. Notice that the same rolls get repeated again and again (just like we warned you they would). Time for a new command!

Scrambling RND With Randomize

Add this to the program and RUN it again:

15 RANDOMIZE

Run it several times and you'll see that your rolls are now different and unpredictable, a much better situation for games! Each time the computer comes across the command **RANDOMIZE** it scatters the "seed" number for the pseudorandom number generator. The end result is that **RND** will produce numbers different from what they would be without **RANDOMIZE**. Since **RND** is a function it can be used in either *deferred* or *immediate* mode; but **RANDOMIZE** is a program command word, and can therefore only be used in *deferred* mode.

Of course, few dice games are played with one die. To make this simulation more real we should add a second die to the throw. Easiest thing in the world:

62 PRINT "AND A ";
63 PRINT INT(6*RND)+1

And to conserve screen display space, **EDIT** line 60 and put a semi-colon on the end. Your program should look like this now:

```
10 REM DICE THROW SIMULATION
15 RANDOMIZE
20 PRINT "HOW MANY ROUNDS?"
30 INPUT N
40 FOR X=1 TO N
50 PRINT "YOU ROLL A ";
60 PRINT INT(6*RND)+1;
62 PRINT "AND A ";
63 PRINT INT(6*RND)+1
70 NEXT X
995 FOR Z=1 TO 300
996 NEXT Z
999 END
```

Button, Button, BTN(n)

In a real dice game the players get to throw the dice for each roll themselves. We can simulate one-at-a-time throws by using another

Applesoft has no **RANDOMIZE** command. The seed is scattered by the length of time a user takes to respond to an **INPUT** or **GET** command.

To read button zero from Applesoft, you must use **PEEK** (-16287) instead of **BTN(0)**; the equivalent of **BTN(1)** is **PEEK(-16286)**. If the button is depressed, then the value returned will be a number between 128 and 255 inclusive.

of HOB's special APPLE ONLY functions, **BTN(n)**. Each game paddle has a button on it; like the function that gets a value for a paddle [**PDL(n)**], the value for (n) can be either a 0 or 1 depending on which button we're talking about, the button on **PDL(0)** or the button on **PDL(1)**.

In immediate mode enter

BTN(1)

HOB will say

THE RESULT IS 0

Now type again (*but don't press [RETURN]*)

BTN(1)

and pick up **PDL(1)**. Press and hold down the button, and press **[RETURN]**:

THE RESULT IS 1

If the button on **PDL(n)** is being held down when the **BTN(n)** function is being looked at by the computer, a 1 will be returned; if the button is not being pressed, a 0 will be returned.

So if we add the following lines the computer will wait for a push of the button before it rolls the dice (notice yet another new command in line 41)

41 BEEP

42 PRINT "YOUR ROLL:"

44 IF BTN(1)=1 THEN 50

46 GOTO 44

The Beeper

To "beep" the Apple's speaker from Applesoft, either print or type **[CTRL] [G]**. This will produce a 1000 Hz tone. A **PEEK (-16336)** will click the speaker once; **POKE -16336,0** will click it twice.

Line 41 makes the computer beep to get the operator's attention (and to add a little life). Actually HOB has three **BEEP** commands, each one producing a slightly different sound. The commands are:

BEEPHI producing a 1778 hertz tone

BEEP producing a 1000 hertz tone

BEEPLO producing a 563 hertz tone

Hertz means cycles per second. Later on you can write your own program to hear what sound each one produces. **HINT:** put a delay loop with a highlimit of about 10 between each **BEEP**-type command.

Line 42 of the program currently in memory displays a message to remind the operator to do something to get things going. Lines 44 and 46 make up a little loop that will spin away until the button on **PDL(1)** is pushed. When that happens, the computer will branch to line 50 and "roll the dice." But players get the feeling that they, rather than the computer, roll the dice since they're the ones who

push the button. Just makes things a bit more personalized.

You can use this same function to take the place of the delay loop at lines 995 and 996. Have a little loop running over and over until a button on one of the paddles is pushed; when it's pushed, let the computer go on to find the END statement. Here we'll use the button on the other paddle [PDL(0)]:

```
995 IF BTN(0)=1 THEN 999
996 GOTO 995
```

RUN

It's better to use this kind of a construct than to use a delay loop. When you use delay loops you have to make assumptions about how fast people can read or how long they want to look at something. Fast readers will get bored waiting and slow readers will get upset because the screen changed before they had a chance to finish reading. When people have control over how long a screen stays displayed, things are better all around.

And to make things even better and more understandable add:

```
35 PRINT "PRESS BUTTON ON PDL(1)";
36 PRINT "TO ROLL DICE"
37 PRINT
```

and

```
68 PRINT
80 PRINT
90 PRINT
100 PRINT "PRESS BUTTON ON PDL(0)";
110 PRINT "TO END"
```

Figure 28 shows what a run of the program should look like now.

A Special Way to Renumber

Before you **ROLLOUT** this program under some appropriate name (we're using the name **DICE**) use the **RENUMBER** command to even up the line numbers a little. But before you do, add this line:

```
900 REM ABS 2000
```

Then enter the **RENUMBER** command. Finally **LIST** the program.

As you can see, all lines below the original line 900 have been renumbered by tens, just as you might expect. But line 900 is now 2000, far higher than you might expect. Take the original line 900 as reading

```
900 REMEMBER: THIS LINE IS ABSOLUTELY
TO BE NUMBERED 2000 WHEN THE
RENUMBER COMMAND IS ISSUED
```

The command is **REM ABS lnum**, where *lnum* is the line number you

want the current line to have. You can have as many of these special renumbering commands as you want in a program. There are only two restrictions: this command can only be used in deferred execution mode, and you can't ask a line to have a number lower than it would be naturally. That means that you can't have the fifth line in a program that is renumbered by tens to have the line number 25!

After you **RENUMBER** your program, you can replace the **REM ABS** line with a more meaningful REM line, for instance

2000 REM PROGRAM ENDING LOOP

FIGURE 28

```

HOW MANY ROUNDS?
?6
PRESS BUTTON ON PDL(1) TO ROLL DICE

YOUR ROLL:
YOU ROLL A 3 AND A 6

YOUR ROLL:
YOU ROLL A 3 AND A 4

YOUR ROLL:
YOU ROLL A 5 AND A 4

YOUR ROLL:

```

The Last Command: DEF FNvar

The penultimate command we'll cover in this chapter is special in that it lets you actually extend the HOBASIC language. The command is called **DEFINE FUNCTION** and it allows you to create new functions.

Let's assume that there's some expression or arithmetic formula that you use over and over in a program. The **DEFINE FUNCTION** command allows you to create that formula once and then use only a few keystrokes to reproduce it.

Begin by clearing the computer with the **NEW** command. Next

type in the following:

```
10 DEF FNF=C*9/5+32
```

This line says: *DEFine a FuNction F such that whenever it is invoked it will return the current value of C times 9/5 with 32 added.* This expression might look familiar; it's the conversion formula for Farenheit to Centigrade temperature. Add the following:

```
20 LET C=100
```

```
30 PRINT FNF
```

```
999 END
```

Now **RUN** the program. When you go back and look at the **PRINT** screen ([**CTRL**][**P**]) you'll see that 212 is displayed. Notice that line 30 says **PRINT FNF**. That's not the variable **F**, but the function **F**. Summon the **SYMBOLS** table. Notice that there's only one variable there, the variable **C**. **FNF** isn't listed because it's not a variable. It's a function, just like **SQR(n)** or **COS(n)**.

In immediate mode enter

```
LET C=200
```

and

```
FNF
```

HOB will respond with

```
THE RESULT IS 392
```

The new *Centrigrade Conversion* function, defined within the program, now operates outside the program in immediate mode. Add this to the program:

```
15 DEF FNC=(F-32)*5/9
```

```
25 LET F=32
```

```
35 PRINT FNC
```

```
RUN
```

Look at the **PRINT** screen; that 0 isn't an error. 32 degrees Farenheit is 0 degrees Centrigrade; the new function created by line 15 does Farenheit Conversion, and it too is now available in immediate mode. Unfortunately, as soon as you type **NEW** we lose the defined functions. They are part of the **HOB** system only temporarily. But you can always recreate them.

You can use an alternate form to create and use a defined function that makes it look more like most other functions (that is, an argument enclosed within parentheses). Clear the machine with **NEW** and enter this code (and see if you can figure out what the command in line 20 does):

```
5 REM CENTIGRADE/FARENHEIT CONVERT
```

```
10 DEF FNF(C)=C*9/5+32
```

Applesoft only uses functions with the form that has an argument. (Eg. If supports **FNX(y)**, but not **FNX**.)


```
15 BEEP
20 HOME
25 PRINT "CENTIGRADE TEMPERATURE:"
30 INPUT C
40 PRINT
50 PRINT C;" DEG. CENTIGRADE IS"
60 PRINT FNF(C); "DEG. FARENHEIT"
70 PRINT
80 PRINT
993 PRINT "PDL(0) BUTTON ENDS"
994 PRINT "PDL(1) BUTTON GOES AGAIN"
995 PRINT
996 IF BTN(0)=1 THEN 999
997 IF BTN(1)=1 THEN 15
998 GOTO 996
999 END
RUN
```

Lines 10, 30 and 60 do the hard core work of the program. Line 10 defines a function called F with an argument called C; line 30 finds out the current value for C; line 60 computes and displays the results. Those are the only lines the computer needs; the rest of the program is for humans (like the HOME command in line 20, which clears the screen).

But defining functions in this new way allows something else to occur. In immediate mode enter

```
LET A=10
FNF(A)
```

HOB will tell you

THE RESULT IS 50

Running this through FINETRACE we get the following:

```
FNF(A)
FNF(10)
(10*9/5+32)
(90/5+32)
(18+32)
(50)
50
```

THE RESULT IS 50

Looking at line 10 where the definition was originally entered and comparing it with what has happened here in immediate, values for A have replaced values for C. Assign values to other variables and try

putting those variables into FNF(var); you'll see that the formula substitution is consistent.

Rules for Defined Functions

1. You can't use a function you haven't defined.
2. You can't redefine a function in the same program with the same name.
3. If you define a function as having an argument [FNF(C) as opposed to FNF] then you can't refer to that function WITHOUT using an argument.

You can define up to 26 functions — one for each letter in the alphabet.

We've covered a lot of material in this chapter. Take time to practice and play with what you've learned. Don't forget to do the exercises at the end.

Chapter Review

Commands:	RND	INT(n)	REM	RANDOMIZE
	BTN(n)	BEEP	BEEPHI	BEEPLO
	HOME			
	REM ABS Inum	DEF FNvar		
	DEF FNvar1(var2)			
Terms:	PSEUDORANDOM NUMBER GENERATOR			
	INTERNAL COMMENTS			
	DEFINED FUNCTIONS			

Practice Exercises

Using the BASIC commands you now know, you can develop the practice dice program you wrote earlier into a real game. You'll need to use IF-THEN commands, subroutines, and in general most of what you've learned. You'll find a version worked out for you in the last appendix in this manual; but don't look at it until you've worked out your own version.

Here are the rules of the game:

1. If the sum of the dice on the first roll is a 7 or 11 the player wins.
2. If the sum of the dice on the first roll is a 2, 3 or 12 the player loses.
3. If the sum of the dice after the first roll is a 7 the player loses.
4. If the first roll is not an immediate win or an immediate loss then the sum of the dice on the first roll is called the player's POINT. Player wins by GETTING HIS POINT on some subsequent roll of the dice (that is, the sum of the dice must equal the player's POINT).



Section Three:

PUTTING HOB TO WORK



CHAPTER 10: PLANNING PROGRAMS

Up to now you've been gathering various programming commands and techniques and you've been practicing them in a number of different ways. Now it's time to take what you've learned and apply it to the development of an actual applications program. An applications program is any program written to solve a particular problem. The problem here is to develop a computerized ordering sheet for a fast-foods operation called *BURGER IN THE SAC*.

Burger in the Sac

BURGER IN THE SAC, or *BITS* as the folk on Wall Street like to call it, is a brand new progressive company whose aim is to have a chain of truly automated fast foods restaurants. Part of that automation is a drive-up order computer. The idea is for customers to drive up to the order computer and to place food orders by operating the computer. Where you come in is helping to write the program that gets orders from customers and prepares their bills.

Program Specifications

Before you can begin to write a program you need program specifications to give you some sort of direction. The specifications tell you what the program should include. Here are the specifications for our part of the *BITS* program:

1. There must be a MENU of five items from which the customer can choose. The menu must show the name of the item and its price.
2. There must be INSTRUCTIONS written in concise language so that people will know how to order their food.
3. The name of the company must be prominently displayed.
4. After an order has been placed the customers must be able to see what they ordered and how much they have to pay.
5. If multiples of any item were ordered (for instance 3 HAMBURGERS or 2 SMALL FRIES) then the total cost of each item must be shown.
6. The items and prices are to be: HAMBURGER—\$1.45; CHEESEBURGER—1.85; LARGE FRIES—.75; SMALL FRIES—.55; ROOT BEER—.60

7. The menu might expand at some future time.
8. After an order has been placed and a bill presented, the MENU must reappear.

Laying Out the Program

All good programs need to be laid out before the programmer hits the keyboard (or the skids, depending on the success of the planner). We'll take the specifications given to us by the *BITS* bigwigs and plan our program from them. Analyzing the requirements tells us that we need to make plans for three major areas: (1) we need to pay attention to WHAT IS DISPLAYED ON THE COMPUTER SCREEN; (2) we need to figure out HOW TO ACCEPT ORDERS; and (3) we need to plan HOW TO COMPUTE THE BILL. We'll do all our planning by determining what the programming phases should look like. This isn't the only way to plan a program, but it's a method that will work fine for the program we're working on.

Program Phases

In general, programs have four phases: *initialization*, *input*, *process*, *output*. The *initialization* section sets up variables and does other tasks to manage the computer's resources so that the program can run most effectively and efficiently. In the *input* phase the computer gets DATA to work on. As you remember from previous chapters such data comes from LET, READ/DATA and INPUT statements. In the *process* stage the computer takes the DATA and works on it: multiplies or divides it, raises it to powers and otherwise performs various operations on it. Finally comes the *output* phase, when the computer takes the processed data and does something with it: prints a report, turns the camera on a VOYAGER spacecraft, or messes up your credit card bill. Quite often there is some processing that occurs during both the *input* and the *output* phases, but while this is still new to you it's useful to think about these phases as being separate and distinct.

We won't know what variables we'll need until we've set up more of a plan for this program so we'll come back to the *initialization* phase later. Instead we'll go right into *input*.

The Input Phase

The *BITS* program must show the customers what choices they have on some sort of a menu display screen. The program must also

tell them how to place their orders, perhaps on the same display. We can think of the menu display screen as an elaborate set of prompt lines for the computer operator (who here happens to be a hungry *BITS* customer). Luckily, our team in the computer display divisions of the company has decided what the menu screen should look like (although they haven't written the code to make it happen; that's our job):

BURGER IN THE SAC	
ITEM	PRICE
1) HAMBURGER	1.45
2) CHEESEBURGER	1.85
3) LARGE FRIES	.75
4) SMALL FRIES	.55
5) ROOT BEER	.60

**PRESS THE NUMBER OF YOUR CHOICE.
THEN PRESS THE RETURN KEY.**

PRESS 0 & RETURN WHEN DONE ORDERING.

YOUR ORDER PLEASE:

Based on the above illustration, we can see that we'll be accepting *inputs* after the screen says *YOUR ORDER PLEASE:*. Now we have to decide what *algorithm* to use for accepting a complete order. An *algorithm* is a step-by-step procedure for solving a problem.

We can't just accept one number as an *input* and then go on. When you go out for a hamburger, chances are you order something else with it. Maybe you order two hamburgers. So we have to have some way of letting customers order as much as they want. We also need a special signal from them to tell us when they're done.

The picture of the menu gives us much of our algorithm. It says the customer must press a number-plus-[RETURN] for each item wanted. The end signal is a [0].

The algorithm looks like this:

1. Present the customer a menu of items.
2. Get an item number from the customer.
3. See if the number given is the "order done" signal (0).
 - A. If it's a 0 then stop accepting orders and go to BILLING.
 - B. If it's not a 0 then continue to the PROCESS phase.

This isn't the complete algorithm, but it's as much as we can get from the picture. We have to add some *process* phase stuff here.

The Process Phase—Take 1

Before we accept another item in this order from the customer (assuming the number entered wasn't 0) we must store the item number ordered someplace. If we don't, we'll lose track of what's been ordered so far and the customer will never get any food. Besides, specification 5 demands we keep track of the quantities of each item sold. We also have to keep track of the customer's bill. So we add the next two steps to the algorithm:

4. Increment the proper item counter.
5. Add the cost of this item to the customer's bill.

Now we can start the process again:

6. Go back to algorithm step 1.

Ordinarily we'd continue with the planning stage for every part of the program; but since this is your first job, we'll stop here and implement the planning we've already done by writing the first part of the program.

Initializations & Inputs

Now we know enough to set up some variables for the program. First, there are the menu items:

```
10 LET H$="HAMBURGER"  
20 LET C$="CHEESEBURGER"  
30 LET L$="LARGE FRIES"  
40 LET S$="SMALL FRIES"  
50 LET R$="ROOT BEER"
```

Next there are the prices for the various items. We'll use an array format, being careful to have the price in each element match a food item listed in the order of the list (that is, the first price will be for a hamburger, the second price will be for a cheeseburger, and so on):

```
55 LET N=5  
60 FOR X=1 TO N  
70 READ P(X)  
80 NEXT X  
90 DATA 1.45,1.85,.75,.55,.60
```

Since we know that the number of items on the menu may change we use the variable *N* to stand for the *number* of items. Later on it will be easy to change *N* to 6, 8, or whatever number represents the current number of items on the menu.

P stands for *price* (since this is the price array). We need a variable to hold the running total of the bill. We'll use *S* for *sum*:

100 LET S=0

We also need a variable to accept *input* of each food item as it's ordered. F for *food*:

1.110 LET F=0

Finally, we need five variables to act as counters to know how many of each item were ordered, a perfect spot for another array. This one will be called C for *counter*, and we'll make sure when we use this array to keep the element numbers consistent with the order we started in the *price* array (we'll count hamburgers in element 1, cheeseburgers in element 2, and so on). For now we'll just initialize the various elements to zero:

120 FOR X=1 TO N

130 LET C(X)=0

140 NEXT X

Showing the Menu

Now that we have the variables set up it's time to write the code to display the menu. We'll take our cue from the menu picture provided for us and write some lines to duplicate it. First we'll make sure the screen is clear (the command **HOME** clears the screen) and we'll write the heading (company name, column headers):

150 HOME

160 PRINT TAB(6);"BURGER IN THE SAC"

170 PRINT

180 PRINT TAB(7);"ITEM";

190 PRINT TAB(18);"PRICE"

200 PRINT

The next five lines each print the item number, a space (just to be neat), the item name; and then tabs over to the next tab field position to print the item's price:

210 PRINT "1) ";H\$,P(1)

220 PRINT "2) ";C\$,P(2)

230 PRINT "3) ";L\$,P(3)

240 PRINT "4) ";S\$,P(4)

250 PRINT "5) ";R\$,P(5)

It now becomes clearer why we needed to make sure that items and prices matched up: item #3, **LARGE FRIES**, matches the price stored in P(3), 75 cents. You'll see more of these matches later. Now come the instructions:

260 PRINT

270 PRINT "PRESS THE NUMBER OF ";

a COMMA used within a **PRINT** statement moves the cursor to the next TAB FIELD position; the positions on the APPLE's 40-column line are 1, 17 and 33

```

280 PRINT "YOUR CHOICE."
290 PRINT "THEN PRESS THE RETURN ";
300 PRINT "KEY."
310 PRINT
320 PRINT "PRESS 0 & RETURN WHEN ";
330 PRINT "DONE ORDERING."
340 PRINT
350 PRINT "YOUR ORDER PLEASE:"

```

Getting The Order

Here's where we apply the *input* algorithm we worked out earlier. Line 360 will get an item number. Line 370 will see if the number input is the *DONE ORDERING* signal; if it is, the computer goes to the *BILLING* section of the program (which will begin at line 410):

```

360 INPUT F
370 IF F=0 THEN 410

```

The Process Phase—Take 2.

Line 380 will take the cost of the particular item chosen and add it to the running total. It says: *Let the variable holding the SUM [S] hold [=] the old SUM [S] plus [+] the PRICE [P] of the FOOD just chosen [F]:*

```

380 LET S=S+P(F)

```

As we said earlier, *process* is often mixed with *input* and *output*, and this line is about as *process* as you can get.

If it weren't for arrays we'd have to write five separate sections of code telling the computer what to do in each of the five possible cases (one for each food item). Look at this demonstration code (but don't type it into your machine). It shows how we might have to handle keeping the running total if we didn't have arrays:

```

380 IF F=1 THEN 1000
390 IF F=2 THEN 2000
(ETC FOR ALL FIVE)
1000 LET S=S+1.45
1010 GOTO 430
2000 LET S=S+1.85
2010 GOTO 430
(ETC FOR ALL FIVE)

```

Back to the real program. Line 390 uses the other array we set up to keep track of how many of each item have been chosen:

390 LET C(F)=C(F)+1

F can be 1,2,3,4 or 5 depending on the food item chosen. Assuming that F were 2 (**CHEESEBURGER**), then variable **C(2)** will hold the old contents of **C(2)** plus 1. If three cheeseburgers had already been ordered [**C(2)=3**] then after line 390, **C(2)** would show that four cheeseburgers were ordered [**C(2)=4**].

Line 400 branches back for the next food order:

400 GOTO 360

And that ends the *input* phase. If any of this is confusing to you, add an **END** line to the program at 9999 and single-step your way through it using the **LIST TRACE** ([L] key) and **CHRONOLOGICAL TRACE** ([T] key) **PROGRAM TRACKING SCREENs** to see how things work. Then come back here and we'll move on to the *output* phase.

The Output Phase

In this section of planning we decide what the *output* from the computer is going to look like and how we'll make the computer show what we want. We need another screen picture and we need another algorithm. The boys in computer display are on top of things again! Here's a diagram that just arrived in interoffice mail:

HERE'S THE BILL

2	CHEESEBURGER	3.70
1	LARGE FRIES	.75
1	SMALL FRIES	.55
2	ROOT BEER	1.20
TOTAL: \$6.20		

**THANKS FOR STOPPING AT
BURGER IN THE SAC
(PLEASE DRIVE TO PICK-UP)**

We see that we need:

1. a heading
2. the quantity, name and cost total for each item ordered
3. a bill total
4. a concluding message with company name

Specifications 1 and 4 are just **PRINT** stuff; no problem. We have already stored #3 in the variable **S** (for **SUM**), so that's just a **PRINT** as well.

Specification 2 will require a bit of *process*. We have the quantities ordered for each item stored in the array called **C**; we set up **C**'s elements to act as counters. We also have the names of the items

stored in various string variables. What we don't have already stored are the cost totals for each item.

That looks easy, though. Cost total per item is just the price of a single item multiplied by the number of items ordered. We have the price per item stored in the **P** array and the quantity per item stored in the **C** array. So all we have to do is to multiply the contents of the appropriate **P** array element times the contents of the appropriate **C** array element to get our cost total.

We have to be a bit careful in how we lay out the code, however. We're only interested in printing out the names and totals for items ordered. Notice that there are no **HAMBURGERS** listed in the diagram; no order, no display! That means that before we do any displaying or processing for an item we need to check its counter to see if anything's been ordered.

Keeping the above warning in mind we can set up the algorithm:

1. Display the heading
2. Look at an element in the counter array
3. If the contents are not zero then
 - a. display the contents
 - b. display the food item's name
 - c. multiply the contents times that item's price
 - d. display the results of the multiplication
4. If there are more elements in the counter array then go back to step 2
5. Print the sum (held in variable **S**)
6. Print the ending message

Steps 2 and 4 present problems: how do we keep track of what array element we're looking at and how do we know when we're done looking? One solution is to use another counter to count the counters! Here's a suggestion on how to implement this algorithm:

```
105 LET I=0
410 HOME
420 PRINT TAB(15);"HERE'S THE BILL"
430 PRINT
440 PRINT
```

Lines 410-440 clear the old text from the screen and set up the new text (so that step 1 of the new algorithm can be carried out). Line 105 sets up the variable **I**. We initialized it at line 105 because it's a good programming practice to initialize all variables in the same section of the program (where they can be easily found). Line 450 starts a

loop to do steps 2,3 and 4 of the algorithm:

```
COUNTER→ 450 LET I= I+1
          460 IF C(I)= 0 THEN 590

STEP 3a→ 470 PRINT C(I),

          480 ON I GOTO 490,510,530,550,570
          490 PRINT H$,
          500 GOTO 580
          510 PRINT C$,
          520 GOTO 580
STEP 3b→ 530 PRINT L$,
          540 GOTO 580
          550 PRINT S$ ,
          560 GOTO 580
          570 PRINT R$,

STEP 3c→ 580 PRINT C(I)*P(I) ←PROCESS STEP
STEP 4→ 590 IF I<N THEN 450
```

The variable *I* can have a value from 1 to 5, representing each of the five food items that can be ordered. It gets incremented in line 450, and line 590 makes sure it isn't more than *N* (set to 5 in line 55). The loop from 450 through 590 gets executed up to 5 times; line 460 doesn't allow lines 470 through 580 to be repeated if the counter array element representing quantity of food items ordered has a 0 in it (no order, no display).

Lines 480 through 570 see that the name of the proper food item is displayed (algorithm step 3-b). We were careful to list the items in their proper order so that what we commanded the computer to PRINT would match the proper value for *I*.

No matter what food item is printed, the computer branches to line 580 so that step 4 can be carried out and display the item cost. On to step 5:

Finishing The OUTPUT

```
600 FOR X=1 TO 6
610 PRINT
620 NEXT X
```

(a quick way to have six blank lines printed)

```
630 PRINT "TOTAL:", "$";S
640 PRINT
```

```
650 PRINT TAB(5);"THANKS FOR STOP";  
660 PRINT "PING AT"  
670 PRINT TAB(7);"BURGER IN THE SAC"  
680 PRINT  
690 PRINT TAB(3);"(PLEASE DRIVE TO";  
700 PRINT " PICK-UP)"
```

All that's left to do is step #8 of the specifications sheet: *After an order has been placed and a bill presented, the MENU must reappear.* We don't want the present screen to disappear, however, before the customer has finished reading it. One solution is using the **BTN(n)** function. We'll ask the people in hardware engineering (their office is down the hall from computer display) to handle the connections; they'll put a game paddle button beneath a steel plate that customers' cars must drive over to get to the PICK-UP window. When the car drives over the plate it will press the button, causing the screen to clear and the program to reset itself:

```
710 IF BTN(1)=1 THEN 100  
720 GOTO 710  
9999 END  
RUN
```

All the variables that were set before line 100 can stay set; we need to use their values again for the next customer. But the other variables (**S**, the counters, etc.) need to be set to zero. If those variables don't get reset they will (of course) retain their old values. If that happens then when the next customer orders, the bill total will include the total for the last order as well as the total for this order (very bad for customer relations). And that assumes the program will run at all; to test this, change line 710 to

```
710 IF BTN(1)=1 THEN 150
```

and **RUN** the program again.

We'll also add a provision so that the manager of a *BITS* franchise will be able to shut the computer down for the night (we'll have **ENGINEERING** tack up a paddle next to the office safe):

```
715 IF BTN(0)=1 THEN 9999
```

And that's that! We've written a program that admirably satisfies the specification sheet. Watch the mail for your check from *BITS*; when they see your program, they'll probably include a bonus.

CHAPTER 11: GETTING THE BUGS OUT

Computers tend to be highly reliable machines. We can, in fact, say that if a computer has no malfunction in any of its parts it *cannot* make mistakes. Unfortunately we can't say the same thing for people. Computers can only do what we as programmers tell them to do. Quite by accident (or at least without intention) we tell computers to do things we really don't want them to do.

As a quick example, take this little program:

```
110 LET J=0
120 LET J=J+1
130 PRINT "EEK! A BUG!"
140 IF K=10 THEN 160
150 GOTO 120
160 END
```

A careful reading will reveal that this program will go on forever! The bug is in line 140. The line should read

```
140 IF J=10 THEN 160
```

The letter **K** has been typed instead of **J**. An easy thing to have happen; **K** is next to **J** on the keyboard and fingers will slip. We didn't mean to tell the computer to test the value of the variable **K**, but we did. The computer can't do what we mean it to do; it can only do what we tell it to do. Planning programs carefully will drastically reduce the number of errors in your coding. But being human, no matter how much planning you do, most of your programs will probably not run flawlessly the first time. Bugs will creep out of the woodwork and into your code. A large part of programming is finding these pesky critters and wiping them out.

Debugging Tools

While most programmers enjoy it up to a point, **DEBUGGING** (the process of finding and correcting programming errors) is not always an easy task. **HOB** provides you with some great tools to make the job simpler and more enjoyable. These tools are not part of the **BASIC** language but are **HOB** special features. At first the real value of these tools may not be apparent to you, and you probably won't make complete use of them immediately. But as you gain more ex-

perience you'll find yourself using the debug tools more and more. You needn't be too concerned if you find yourself saying "So what?" to one or more of them. For now it will be enough for you to follow through the chapter experiencing the tools so that you'll have a feel for them.

NOTE: This chapter makes extensive use of the dice game program listed at the end of Appendix J. If you haven't already stored it on your ROLLIN-ROLLOUT diskette we recommend you type that program into your computer's memory and store it before continuing.

You've already seen and used many of HOB's debugging tools. You've used a number of PROGRAM TRACKING SCREENs, certain immediate commands like PACE and LIST, and you know how to see the values of variables by summoning them individually (by entering a variable name and pressing [RETURN]) and collectively (via the SYMBOLS table). This chapter will introduce you to the remainder of HOB's debugging aids.

Using the Appendices

You can make most effective use of these aids by referring to the appendices in the back of this manual. Appendix A, IMMEDIATE COMMANDS, for example contains a synopsis of all the immediate commands in HOB. These commands are often extremely useful in debugging. Appendix D, ERRORS AND POSSIBLE SOLUTIONS, explains all the error messages HOB will deliver when it discovers a bug, and includes a number of suggestions about what might have caused a particular error and how you can correct it. Appendix E, PROGRAM TRACKING SCREENs, sums up HOB's SCREENs and reminds you what keys to press to access them. And you can make good use of Appendix H, EDITING KEYS AND COMMANDS, to save you excess typing during program debugging.

The Variables Screen

As we've already mentioned, among HOB's debugging tools are the PROGRAM TRACKING SCREENs. You've been using seven of the eight available screens: COMMAND, CHRONOLOGICAL TRACE, FORLOOP, LIST TRACE, DATA, GOSUB and PRINT (see Appendix E for a brief recap of each).

The final screen is VARIABLES. It shows the current values for all variables in a program and the line numbers in which the values changed.

Actually, you've seen VARIABLES before in another guise. VARIABLES is like the SYMBOLS table (introduced in Chapter 2) gone dynamic. While you could view SYMBOLS only when program execution had ended or had been interrupted via [ESC] or an error, you

can call for VARIABLES at any time during program execution by pressing the [V] key.

To demonstrate how to use the VARIABLES screen effectively, you'll need a fairly large program loaded into your computer's memory. We'll use the **DICE GAME PROGRAM** listed at the end of Appendix J. (Full rules for the game are listed in the PRACTICE EXERCISE section at the end of Chapter 9.) We'll continue to use this program to demonstrate the rest of the tools in this chapter.

Three of THE DICE GAME's variables are particularly important: **R1** and **R2** hold the numbers rolled on each of the two dice; **P** holds the POINT the player is trying to roll (and just to refresh your memory, 7 or 11 on the first roll wins; 2, 3 or 12 on the first roll loses; 7 on any subsequent roll loses).

RUN the program. We'll assume here that you've played this game several times before and are familiar with how it works and what it does; if that's not so, take some time now to learn the program. After you've entered your name press the [V] key. Your screen should look like figure 29.

The number to the left of the variable name is the line number in which the current value for the variable was set. The number to the right is the variable's current value. **R1** currently equals 0 and was set in line 8.

```

1030  27888 FULL

70 N$      ="PAUL"
4 P        =0
8 R1       =0
9 R2       =0
6 T        =0

5/5

```

FIGURE 29

By the way, the numbers 1020 and 1030 are flashing like crazy at the upper left of the screen because the program is “spinning its wheels” between those two lines. Press [L] to switch to the LTRACE screen to see the operation “in context” with the rest of the lines in this subroutine. Then press the [V] key to get back to VARIABLES.

Press PDL(1)’s button now to make the values of the variables change. The button has nothing to do with the VARIABLES screen; pressing it “rolls the dice” in the program we’re running. Not only will the values for the variables change, but so will the line numbers in which the changes occur (numbers to the left of the variable names). R1 gets its current value from line 1040 (whatever that value is — ours is 4 but yours could be any number between 1 and 6).

Continue pressing PDL(1)’s button until the game ends (which will happen when $P=R1+R2$ or when the sum of R1 and R2 is 7).

“Paging” Through the Variables Screen

If there were more variables defined in DICE than would fit on one screen page (only eight will fit), HOB would keep tabs on them by putting them on additional pages. You could look at these other variables by “flipping through the pages” (eight variables at a time) using the same commands you were shown under SYMBOLS. In case you’ve forgotten: To see the next eight variables you would press the [+] key; you would continue pressing the [+] to see more. To “back up” to the previous eight variables shown, you press (what else!) the [−] key. At any point you could go back to the first “page” of variables by pressing the [@] key.

Summed up, the commands are:

[+]	look at next eight variables
[−]	look at previous eight variables
[@]	look at first eight variables

STOP! If you have just typed in Appendix J’s dice game program and have not yet stored it onto a diskette do so now! Otherwise it will be eaten by the program coming up next!

In an earlier chapter we summoned a hidden program from deep within HOB. Even as we speak there’s another one roaming around the dark recesses of your computer’s memory chips that would serve us well now. Call it up by entering

#S

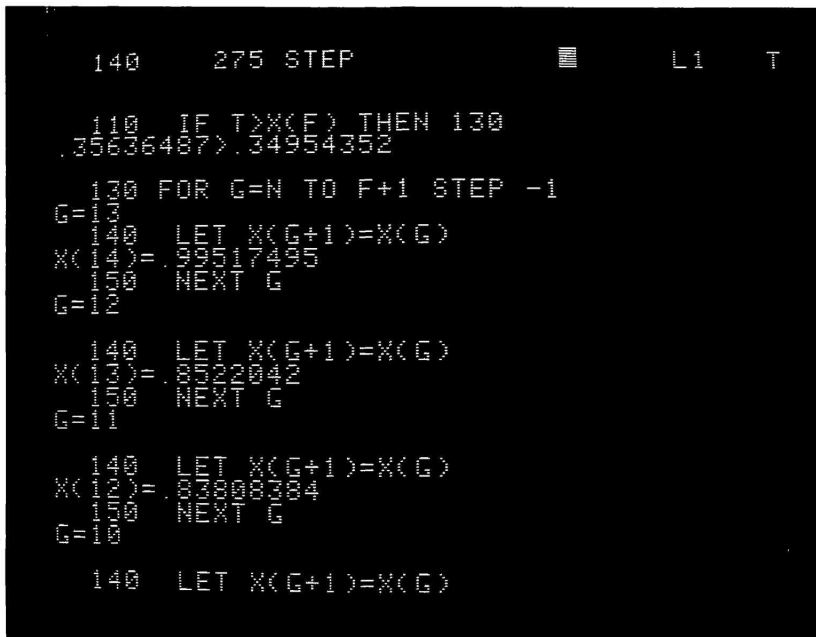
RUN this new program and press [V] to switch to the VARIABLES screen (there aren’t many PRINT statements in the program; not much will show on the PRINT screen). There you’ll see the values of

numerous variables changing before your eyes. Page through the variables by pressing the [+] key.

Checking Values of Compared Variables

Switch to the CHRONOLOGICAL TRACE screen (press [T]) and get into SINGLE STEP mode (press the [SPACE] bar). Skip through the program a bit via the [←] and [→] keys until you get something like figure 11-B on your computer's display screen. (Another way to get to the same place is to press [ESC] and enter RUN; immediately press [T]; watch the PROGRAM COMMAND COUNTER—second number from the left at the screen top; when it gets to about 275 press the [SPACE] bar.)

You learned earlier that CTRACE, along with a chronological listing of the program-in-progress, displayed variables and their values as they changed. CTRACE also shows the values of compared variables. In figure 11-B, for instance, line 110 is displayed comparing the values of T and X(F) in an IF/THEN construct. Below the display of line 110 you see the values of both T (194) and X(F) (varying between 198 and 196).



```

140      275 STEP          L1  T

110  IF T>X(F) THEN 130
.35636487>.34954352

130  FOR G=N TO F+1 STEP -1
G=13
140    LET X(G+1)=X(G)
X(14)=.99517495
150    NEXT G
G=12

140    LET X(G+1)=X(G)
X(13)=.8522042
150    NEXT G
G=11

140    LET X(G+1)=X(G)
X(12)=.83808384
150    NEXT G
G=10

140    LET X(G+1)=X(G)

```

FIGURE 30

Comparisons in DO WHILE and DO UNTIL constructs are also displayed, but in a different way. Press [ESC] to stop the program; now enter

#L

This program (obviously) demonstrates *DOLOOPS*. Set **PACE** to **STEP** and **RUN** the program. Press [T] and single-step through the first 10 commands. This loop, initialized in line 30, will continue until N, incremented in line 40, reaches 10. The loop ends at line 50. Each time line 50 is displayed, the values on both sides of the = sign from line 30 are displayed. Notice that the values are actually compared at line 50 in the LOOP command, and not back at line 30. (If they were compared at line 30, then line 30 would be displayed again). Single-step through more commands until the loop is completed (the information below line 50 will read 10=10 and line 60 will appear). Then look at the *FORLOOP* screen; values compared by relational operators for DO WHILE and DO UNTIL constructs are shown there.

Some Final Words on Program Tracking Screens

All PROGRAM TRACKING SCREENs are available for study both during program execution and when execution has halted. As you already know, during program execution you summon a particular screen by pressing that screen's letter key. Summed up, they are:

COMMAND	C
CTRACE	T
DATA	D
FORLOOP	F
GOSUB	G
LTRACE	L
PRINT	P
VARIABLES	V

When execution is interrupted or when the program has completed its run, you may access the screens by pressing and holding down the [CTRL] key before pressing the usual screen key. But if you change any line of program code none of the screens will be accessible until the program is run again. The control sequence also works if the computer is waiting for a response to an *input*.

The Find Commands

FIND allows you to discover all places within a program where a particular line number or variable is referenced or where the value of a variable is either set or changed. **FIND** has three forms, all of which operate in immediate mode only. The first form

FIND *Inum*

(*Inum* is short for *line number*) will immediately display all program lines in which *Inum* is referenced. **ROLLIN** the DICE GAME program and enter

FIND 6000

from the COMMAND screen. Lines 130, 140, 150 and 220 should be listed with the number 6000 displayed in inverse video. These are all the lines in the program that make any reference to line 6000.

The second command in this series

FIND *var*

acts similarly to **FIND *Inum*** in that it will list all lines in which the variable *var* appears. The variable named can be string, numeric, or array. Enter

FIND N\$

and HOB will list 70, 75, 80, 170, 5010 and 6010. N\$ will appear in inverse video. If all these lines don't appear, you haven't typed them into your program.

There's a variation on this command, called

FIND *var*()

which will find and display all instances of the named array variable. Assuming this program contained lines using the one-dimensional array *X* (as in *X(3)* or *X(N)*), invoking the command

FIND *X*()

would cause all references to the array to be listed. Similarly if there were a two-dimensional array called *J* (as in *J(A,B)*), you could see all lines containing a reference to it by entering

FIND *J*(,)

The final command in this series

FIND *var*=

differs from **FIND *var*** in that it will list only those lines in which the value for *var* is either set or changed. Enter

FIND N\$=

and only line 70 will be listed. This is the only line in which the value for N\$ is set or changed. For contrast, enter

FIND T

and

FIND T=

The first command lists eleven lines; the second lists two. Out of the eleven lines referencing T, its value is changed in both lines 6 and 1050.

FIND var= also works for array variables. And as you might expect, the forms are

FIND var()=

for one-dimensional arrays and

FIND var(,)=

for two-dimensional arrays.

The Break Commands

Quite often in program debugging it is valuable to stop program execution to examine the value of specific variables or to trace program flow. HOB's **BREAK** commands allow you complete control over setting **BREAKPOINTS**, places in the program where execution will halt temporarily. **BREAKPOINTS** do automatically what up to now you've had to do manually by pressing the [SPACE] bar or [ESC] key. If the computer encounters a **BREAK** command while you are watching the **PRINT** screen, execution is halted and you are returned to the **COMMAND** screen. If you are watching any of the other **PROGRAM TRACKING SCREENS**, you'll go into single-step mode. There are seven **BREAK** commands.

Three of the **BREAK** commands can be thought of as **FIND** commands that set breakpoints. Those commands are **BREAKFIND var**, **BREAKFIND var=**, and **BREAKFIND lnum**.

BREAKFIND var

Issue the command

BREAKFIND N\$

and the same list of program lines will appear as appeared earlier when you told HOB to

FIND N\$

The difference here is that the program will be interrupted temporarily (and wait for you to press [RETURN] to continue execution) at lines 70, 75, 80 and so on. **RUN** the program to see what happens. When the program stops running and returns to the **COMMAND** screen, the message

BREAKPOINT

will appear at the top left of the screen. You can now switch around among the various screens, summon the SYMBOLS table, examine the contents of a particular variable, and in general check out the condition of the program. When you are ready to resume execution, press [RETURN] and the program will continue until the next breakpoint is encountered.

BREAKFIND lnum

Enter the command

BREAKFIND 1000

and the two lines that reference line 1000 are displayed and are added to the list of breakpoints.

BREAKFIND var=

Enter

BREAKFIND P=

and lines 6 and 1050 are displayed and added to the breakpoint list.

The commands **BREAKFIND var()**, **BREAKFIND var()=**, **BREAKFIND var(,)** and **BREAKFIND var(,)=** do the same thing for one and two dimensional arrays as their non-array variable BREAKPOINT counterparts do. If, for example, you entered the command

BREAKFIND X()

all instances of the one dimensional array **X** would be marked as breakpoints (assuming, of course, there was an array called **X** in the program!).

The fourth BREAK command will let you see all the lines where breakpoints have been set. Enter the command

BREAKLIST

and all breakpoint lines will list. Notice that this list appears with nothing in inverse video. Like a regular listing, you may interrupt a BREAKLIST by pressing the [SPACE] bar; pressing it again causes BREAKLIST to LIST one line at a time. [RETURN] causes it to continue at full speed, and [ESC] will abandon a BREAKLIST.

You can add breakpoints one at a time with the fifth BREAK command

BREAK lnum

For example, enter the command

BREAK 9995

and 9995 will be added to the list of breakpoints.

The final two BREAK commands allow you to undo what you've done. To turn off a particular breakpoint the command

NOBREAK Inum

is used; and to completely clear the breakpoint list, issue the command

NOBREAK

The BREAKPOINT Marker

A final note on the BREAK series: any BREAKPOINTS you've set in a program will be noted on the LIST TRACE SCREEN as a crosshatch (#) between the line number and the body of the command. For instance:

```
150#FOR Z=1 TO 100
```

When the BREAKPOINT has been cleared, the crosshatch will no longer be visible.

Time Machine Revisited— LOOPSTEP & GOSUBSTEP

Quite often in using HOB's PROGRAM TRACKING SCREENs you'll be slowing down program execution either through use of PACE= or by pressing the [←] and [→] keys. You're bound to get into certain situations where you enter some horribly repetitious loop you really don't want to crawl through at 1/8th normal speed. For such times HOB provides the LOOPSTEP and GOSUBSTEP commands.

LOOPSTEP

LOOPSTEP, which is summoned by pressing the [;] key, sets the program to full speed until the termination of the current FORLOOP or DOLOOP; then it goes into single-step mode. For instance, with PACE set to 1 the following program segment will take something like 15 minutes to complete:

```
10 LET J=0
•
•
135 FOR Z=1 TO 1000
140 LET J=J+Z
145 NEXT Z
150 LET ZQ=J
•
•
999 END
```


But it will take far less time if you invoke **LOOPSTEP** by pressing the semicolon; **PACE** will increase to full until line 150. Pressing the [;] when the program is not in a loop will cause the program to run at full speed until the end of the next loop; then **PACE** is set to **STEP**. Play with this command for a while until you get the hang of it.

GOSUBSTEP

The **GOSUBSTEP** command, summoned by pressing the [,] key, does for subroutines what **LOOPSTEP** does for loops. Everything just said about **LOOPSTEP** goes for **GOSUBSTEP**; just substitute **SUBROUTINE** for **LOOP**.

LOOPSTEP and **GOSUBSTEP** are abandoned by pressing [SPACE] bar, [RETURN] and either the [←] or [→] key, or when a **BREAKPOINT** has been reached. To both abandon **LOOPSTEP** or **GOSUBSTEP** and to resume **FULL** speed operation, press [X] or [RETURN].

The Printer Commands

These commands allow you to print on paper any or all of the following: a listing of your program, a **CHRONOLOGICAL TRACE** of your program, a copy of what would appear on the **PRINT** screen throughout a program **RUN**, and a picture of what appears on the computer screen at the time the command is invoked.

The first three commands are respectively **LLIST**, **TR#n**, and **PR#n**. The variable **n** is the slot number in the computer where the connection to the printer has been made. For demonstration purposes, we'll assume that your printer is connected to slot #1; that's the slot most people use. The last command is [S] or [CTRL][S]; it sends to your printer whatever is currently being displayed on your computer's screen.

Program Listings

As your skill in programming grows, so will the size and complexity of your programs. It becomes more and more difficult to develop and debug programs when all you can see are twenty or so lines at a time. Having programs listed on paper where you can have access to all parts of the program at once makes debugging a whole lot easier. Make sure your printer is turned on and enter the following command:

LLIST

WARNING: The following four commands can be used only if you have a printer hooked up to your computer. If you attempt to use any of them without having a printer hooked up, **HOB** will "hang" and you'll have to press [RESET] and start from the beginning. Any program in the computer's memory not **ROLLED OUT** will be lost.

NOTE: If your printer is installed in other than slot #1 then you must enter the following command before you use **LLIST** for the first time:

PRINTER IS IN SLOT n
Otherwise your computer will "hang".

(What you see is not a typo; there really are two L's there) and your program will be listed on paper. Every time you type the **LLIST** command, you will get a hard copy of your program. To interrupt a printout press the [SPACE] bar; press [RETURN] to continue. Press [ESC] to abandon a listing altogether.

Chronological Trace on Paper

Several times in this manual we've referred to the **CHRONOLOGICAL TRACE PROGRAM TRACKING SCREEN** (accessed by pressing the [T] key) to follow the progress of some program. As you've discovered by now programs move along quite quickly; it's often quite a chore to follow the program flow, even if you slow things down via **PACE** or [←]. The command

TR#n

sets things up so that the next time the program in memory is run a chronological trace of the program is printed out.

You'll see that a printer **CTRACE** takes quite a bit longer than does a normal **CTRACE**. The apparent slowdown in program speed happens because the computer has to wait for the printer. The computer displays characters on your screen at a rate of about 2000 characters a second; it operates much faster than that when it's just processing and printing nothing to the screen. Most personal printers, on the other hand, print their text at a rate between 40 and 80 characters per second. Under the best conditions, then, **TR#n** will make your program run at about 1/40th its normal speed. But since you're still debugging the program, that should cause no problem; when *end users* (non-programming operators) run your program it will operate at top speed again.

We got a chronological trace of **DICE GAME PROGRAM** on paper by entering the following:

TR#1
RUN

When the trace finished we typed in

TR#0

If we hadn't, every run of the program would have caused another hard copy of chronological trace. **TR#n** can be aborted at any time by pressing [ESC] followed by the **TR#0** command. Pressing the [SPACE] bar will freeze program execution, which in turn will interrupt printout; both program execution and the printout are continued by pressing [RETURN].

Printing to the Printer

The third command in the series is **PR#n**, and it causes all PRINT statements to be printed on paper as well as displayed on the computer screen. Tabs, commas, semicolons and other commands that affect how text is displayed on the screen also affect what's printed on paper. So you can think of **PR#n** making happen on paper what usually happens only on the screen. You use **PR#n** exactly the same way you use **TR#n**. We got copies of DICE GAME PROGRAM's screen by entering the following sequence:

```
PR#1  
RUN
```

Later we typed

```
PR#0
```

so we could run the program again without having our screens printed out.

Screen Snapshots

Pressing the [S] key during program execution (or [CTRL][S], when execution is halted or when an INPUT prompt is on the screen) will send a "snapshot" of the screen out to the printer; that is, an exact image of what's on the screen will be printed on paper. This very useful command lets you get precise copies of all the PROGRAM TRACKING SCREENs, selected sections of program listings, a print-out of BREAKLIST, and anything else you can see on any of HOB's screens.

For instance, assume you have a program with sixty variables and lots of bugs (groan). Having a list of those variables and their values would make debugging a whole lot easier. By summoning SYM-BOLS and alternately pressing [CTRL][S] to print the screen and [+] to summon the next "page" of variables, you would get hard copy of all your variables.

Used in combination with the other debugging tools, the screen snapshot command can save you hours of debugging time and provide you with some very complete program *documentation* (written explanation of how a program functions).

NOTE: The following command will work as described if your printer is hooked up to your computer through slot #1. If it is hooked up via any other slot, then either one of the other slot-setting printer commands must have been used already or the following command must be entered:

PRINTER IS IN SLOT n
where n is the slot number.

Audio Feedback: The N Key

Sometimes during program execution it's hard to know what's going on inside your computer. If your program has no PRINT com-

mands and no BEEP commands, you may not know if your program is running at all! Faced with a blank screen, visions of a hung system may speed through your fear-filled mind. HOB has an anti-paranoid feature called NOISE. During program execution pressing the [N] key will cause your computer's built-in speaker to click once for every other command executed. The tone of the click will vary depending on how fast your program is running, whether HOB must print anything on a screen (including the PROGRAM TRACKING SCREENs) and a variety of technical reasons. Pressing [N] a second time turns the noise off. Enter this little three-liner and experiment:

```
10 LET A=A+1
20 GOTO 10
30 END
```

To find the amount of memory remaining for an Applesoft program, PRINT the value of FRE(0).

While computers may seem like remarkable tools, they do have their limits. One of those limits is PROGRAM SIZE, the number of memory cells (called BYTES) that a program may occupy. To find out how many bytes the program you currently have in memory takes up (and to see how many more bytes you can add to it), enter the command

SIZE

The Final Command

The final command in the HOB system is an immediate execution one, and it causes HOB to go the way of all flesh. Entering the command

EXIT

returns your computer to its resident language (either Integer or Applesoft BASIC).

The Final Challenge

We've used three of HOB's hidden demonstration programs. There are two more, but you'll have to find them yourself. The only hint we'll give you is that HOB's SYNTAX CHECKER can tell you how to access them. Happy hunting!

The Final Remarks

Throughout this manual we've been stressing the importance of doing rather than just reading. Programming is not an art to be

learned on a theoretical level. The only way to learn to program is to do it. At first your programs will have more bugs in them than a condemned building on skid row. But the more you program the fewer errors you'll make and the more creative you'll become.

Those of us who are into computers know how much joy there is in programming, and how amazingly fulfilling and creative an experience it can be. You can have those experiences yourself in a remarkably short period of time; all you have to do is stay with it.

So code away! And for heaven's sake, pay your electricity bills on time!

APPENDICES

A: Immediate Commands

Following is a brief description of HOB's various system commands. These commands are entered in *immediate mode* only and are used to **execute**, **edit** and **debug** computer programs and *immediate mode* computations. Most single-letter commands are not listed here; see Appendix K.

BREAK Inum program execution will temporarily halt when **Inum** is encountered, allowing examination of the various PROGRAM TRACKING SCREENs and the SYMBOLS table; execution resumes when [RETURN] is pressed.

BREAK Inum

BREAKFIND Inum immediately displays all program lines in which **Inum** is referenced (with **Inum** in inverse video); adds displayed lines to BREAK list.

BREAKFIND Inum

BREAKFIND var immediately displays all program lines in which **var** is referenced (with **var** in inverse video); adds displayed lines to BREAK list.

BREAKFIND var

BREAKFIND var() same as **BREAKFIND var** for one dimensional array variables.

BREAKFIND var ()

BREAKFIND var(,) same as **BREAKFIND var** for two dimensional array variables.

BREAKFIND var (,)

BREAKFIND var= immediately displays all program lines in which the value of **var** is either set or changed (displays **var** in inverse video); adds displayed lines to BREAK list.

BREAKFIND var=

BREAKFIND var()= same as **BREAKFIND var=** for one dimensional array variables.

BREAKFIND var ()=

BREAKFIND var(,)= same as **BREAKFIND var=** for two dimensional array variables.

BREAKFIND var (,)=

BREAKLIST sequentially lists all lines for which breakpoints have been set (i.e. all lines at which program execution will be temporarily halted; see **BREAK Inum**).

BREAKLIST

CATALOG displays listing of all programs currently stored on ROLLIN-ROLLOUT diskette.

CATALOG

DEGREES causes system to treat arguments of trigonometric functions as being expressed in degrees. System default is **RADIANS**.

DEGREES

DEL Inum delete **Inum** from program.

DEL Inum

DEL Inum1,Inum2 delete range of lines **Inum1** to **Inum2** (inclusive) from program.

DEL Inum 1, Inum 2

DELETE (program name)	DELETE <program name> causes <program name> to be removed from ROLLIN-ROLLOUT diskette. A deleted program cannot be recovered.
EDIT Inum	EDIT Inum list Inum to screen with cursor positioned after last character on line, accept changes made to line through use of various EDIT keys.
EXIT	EXIT clear HOB from computer's memory; leave computer in resident language (Integer or Applesoft BASIC).
FIND Inum	FIND Inum immediately displays all program lines in which Inum is referenced; Inum is displayed in inverse video.
FIND var	FIND var immediately displays all program lines in which var appears; var is displayed in inverse video.
FIND var()	FIND var() same as FIND var for one dimensional arrays.
FIND (,)	FIND var(,) same as FIND var for two dimensional arrays.
FIND =	FIND var= immediately displays all program lines in which the value of var is either set or changed; var is displayed in inverse video.
FIND ()=	FIND var()= same as FIND var= for one dimensional arrays.
FIND (,)=	FIND var(,)= same as FIND var= for two dimensional arrays.
FINETRACE	FINETRACE displays step by step computation process used by computer to solve arithmetic problems and formulae entered in immediate mode; each press of the [SPACE] bar displays the next step in the computation process. FINETRACE cancelled through NOFINETRACE command.
LET var=exp	LET var=exp assigns value defined by exp to var. exp must correspond to var type (i.e. numeric variables will not accept strings).
LIST	LIST sequentially displays all program lines; pressing [SPACE] bar during listing temporarily halts list; pressing [SPACE] bar again causes next line to list; [RETURN] continues listing at full speed; [ESC] terminates listing.
LLIST	LLIST (only for systems with printers) sends program listing to printer. Printer is assumed to be plugged into slot #1 unless otherwise specified by PRINTER IS IN SLOT n command (also in this appendix). Pressing the [SPACE] bar interrupts listing; pressing it again causes listing to continue.
	Inum same as DEL Inum .
LOCK (program name)	LOCK <program name> causes <program name> to be protected from accidental deletion. Asterisk (*) appears to left of <program name> when diskette is cataloged. Protection cancelled through use of UNLOCK command.
NEW	NEW clears program from computer memory; clears all PROGRAM

TRACKING SCREENS; clears **SYMBOLS** table; clears **BREAK** list; resets **PACE** to **FULL**.

NOBREAK clears **BREAKLIST** so that no breakpoints are set.

NOBREAK

NOBREAK lnum turns off breakpoint for **lnum** no matter which **BREAK** command originally set the breakpoint.

NOBREAK lnum

NOFINETRACE turns off **FINETRACE** function.

NOFINETRACE

PACE=n sets speed of program execution from 0-5 with 0=single step, 5=full speed, 1=1 command/second; set to 5 by pressing **[RETURN]** or **[→]** several times during program execution. The commands **PACE=STEP** mean same as **PACE=0**; **PACE=FULL** means **PACE=5**. Reset to 5 by **RUN**, preserved by **PUN**.

PACE=n

PR#n (only for systems with printers) causes all text displayed by **PRINT** and related formatting commands to be sent to a printer plugged into slot #n; turned off by **PR#0**. Since command operates during program run it can be interrupted by **[ESC]**, continued by **[RETURN]**.

PR#n

PUN run program in memory without changing current **PACE=** setting.

PUN

PRINTER IS IN SLOT n sometimes needed for the **[S]** screen printout and **LLIST** commands: sets n to slot number through which printer is hooked to system; needed only if printer is not hooked through slot #1 and if no previous printer commands have been issued since **HOB** was loaded into computer.

PRINTER IS IN SLOT n

RADIANS causes system to assume that arguments of trigonometric functions are expressed in radians. **RADIANS** is system default.

RADIANS

RENUMBER base, increment changes all line numbers of program in memory. First line number is **base**, gap between line numbers is **increment**. Defaults are 10 for **base**, 10 for **increment**.

RENUMBER base, increment

ROLLIN <program name> recall program called **<name>** from diskette and place in computer's memory. Program is rolled in in exactly the state in which it was rolled out; if rolled out in suspended state, pressing **[RETURN]** will resume program execution.

ROLLIN (program name)

ROLLOUT <program name> store program called **<name>** plus all related **PROGRAM TRACKING SCREENS** to diskette (see **ROLLIN**).

ROLLOUT (program name)

RUN clear **SYMBOLS** table (see **SYMBOLS**), check branching commands to be sure targets exist, check **FOR-NEXT** loop structures, check existence of **END** statement, reset pseudorandom number generator and **DATA** pointer, begin program execution.

RUN

S (only for systems with printers) sends screen image to printer; use **S**

[CTRL][S] to send screen image when program execution is interrupted or when INPUT PROMPT is on screen.

SIZE **SIZE** displays on screen number of bytes used by program and number of bytes available.

SYMBOLS **SYMBOLS** summons display of SYMBOLS table (same as VARIABLES) which keeps track of current value of all variables (up to 255 numeric and 26 string). Alternate method to [CTRL][V] for summoning VARIABLES while program execution is halted.

TR#n **TR#n** (only for systems with printers) causes RUN of program to send CHRONOLOGICAL TRACE of program to printer plugged into slot #n. [ESC] interrupts printout; [RETURN] continues. Cancelled by **TR#n**.

UNLOCK (program name) **UNLOCK <program name>** cancels effect of **LOCK** command by removing deletion protection from named program (see **LOCK**).

B: Deferred Commands

The following BASIC commands can be used only in *deferred execution mode*; that is, they will operate only from within a program and must be preceded by a line number. Each entry is followed by an example of its proper use within brackets and a brief definition.

Some Shorthand Used In These Definitions

exp—expression. Can be any number, variable or string.

int—integer. Any whole number.

lnum—line number. Any integer between 1 and 9999.

relational—one of the relational operator sets (<, =, >, <>, >=, <=)

var—variable. Either string or numeric unless qualified.

The Command List

BEEP [BEEP] sounds APPLE's beeper with 1000 hertz tone.

BEEPHI [BEEPHI] sounds APPLE's beeper with 1778 hertz tone.

BEEPLO [BEEPLO] sounds APPLE's beeper with 562.5 hertz tone.

DATA [DATA exp, "exp", ...] provides list of items to be used by READ. Also called DATA LISTS. Items may be numbers or strings. Each item must be separated from the next item in the list by a comma. Quotes around strings are optional. DATA lists may appear anywhere in the program. Up to 22 DATA lists are allowed.

DEF FN [DEF FNnumvar=numexp] define a function such that when *FNnumvar* is referenced *FNnumvar* will return a value equal to current value of *numexp*.

DEF FN () [DEF FNnumvar(numvar)=numexp] same as **DEF FN** except will return current value of *numexp* based on any *numvar* and not just that originally defined. For example: **DEF FNA(X)=6*X/5** is defining statement. Later **FNA(Q)** will yield $6*Q/5$ based on current value of *Q*.

DIM [DIM A(int)] or [DIM B(int1,int2)] establishes highest number of elements allowed in an array. If not set **DIM** is assumed to be (10) for one dimensional arrays and (10,10) for two dimensional arrays. One dimensional arrays may be set as high as (4095); two dimensional arrays may be DIM'd to (255,255). Array name may be any single alphabetic.

DO [DO] begins execution of series of commands ending in command

BEEP

BEEPHI

BEEPLO

DATA

DEF FN

DEF FN ()

DIM

DO

LOOP (see **LOOP**); commands between **DO** and **LOOP** commands will repeat infinitely.

DO UNTIL **DO UNTIL** [**DO UNTIL** exp1 relational exp2] begins execution of series of commands ending in command **LOOP**; all commands in series will repeat until the stated relational in the **DO** command line is met. When that occurs execution branches to the command line following the corresponding **LOOP** command.

DO WHILE **DO WHILE** [**DO WHILE** exp1 relational exp2] begins execution of series of commands ending in command **LOOP**; all commands in series will repeat while the stated relational in the **DO** command line is met. When the relational is no longer met execution branches to the command line following the corresponding **LOOP** command.

END **END** [**END**] tells computer to terminate program execution and return to **COMMAND** screen. Must be highest (final) line number in program.

FOR-TO-STEP **FOR-TO-STEP** [**FOR** numvar=numexp1 **TO** numexp2 **STEP** numexp3] used in conjunction with **NEXT** (see **NEXT**). Causes lines between itself and **NEXT** statement to be executed *N* number of times where *N* is the absolute integer of (numexp2–numexp1)/numexp3. **STEP** is optional and is assumed to be +1 unless otherwise stated. If numexp1 is higher than numexp2 then **STEP** must be a minus number.

GOSUB **GOSUB** [**GOSUB** lnum] causes unconditional branch to line number specified. Line number is first line of subroutine ending with the command **RETURN** (see **RETURN**). Up to 16 **GOSUB**s may be nested. Exceeding 16 nested **GOSUB**s causes **OVERFLOW** message to be displayed; program aborts. There is no upper limit to the number of non-nested **GOSUB**s that may appear in a program.

GOTO **GOTO** [**GOTO** lnum] causes unconditional branch to line number specified. Target line number must exist to avoid program abort. **GOTO** into body of **FOR** loop not allowed. May also appear as **GO TO**.

HOME **HOME** [**HOME**] clear **PRINT** screen; move cursor to upper left corner.

IF-THEN **IF-THEN** [**IF** exp relational exp **THEN** lnum] causes branch to line number following word **THEN** if stipulated condition is met. Test of condition is comparison between two expressions using relationals (as in *IF A<>60 THEN 500*).

INPUT **INPUT** accepts data from keyboard (from computer operator rather than from within program). Displays a question mark (?) as prompt to operator. Data entered must correspond to variable type specified by **INPUT** command (i.e. **INPUT A** will reject attempts to enter a string). Computer will wait indefinitely for response from operator.

LET [LET var=exp] assigns value defined by *exp* to *var*. *exp* must correspond to *var* type (i.e. numeric variables will not accept strings).

LET

LOOP [LOOP] causes unconditional branch to corresponding DO command (see **DO**, **DO UNTIL** and **DO WHILE**).

LOOP

NEXT [NEXT intvar] last command of corresponding FORLOOP. Increments *index* of FORLOOP. If *index* is within range of **FOR** command program branches to line following **FOR** statement. If *index* exceeds range of **FOR** statement program execution continues to line following **NEXT** statement.

NEXT

ON-GOTO [ON numvar GOTO lnum1,lnum2, ...] unconditionally branches to *lnum* which is *numvar* positions beyond word **GOTO**. Positions are separated by commas. If command is *ON X GOTO 250,475,300,36* and *X* holds 4 then the branch is to line 36; if *X* holds 2 then branch is to line 475; etc. *numvar* must be between 1 and total number of line references listed; otherwise program aborts with *ON INDEX numvar IS OUT OF RANGE*.

ON-GOTO

OPTION BASE [OPTION BASE int] defines lowest array element number. Variable *int* may be 0 or 1. Defaults to 0. If used must be lowest numbered line in program.

OPTION BASE

PRINT [PRINT] or [PRINT exp;exp, ...] causes the specified expression, tab specification or blank line to be displayed on the PRINT screen. If **PR#1** has been specified then any PRINT data will also be sent to the printer. Items separated by commas will be displayed in different TAB fields. Items separated by semicolons will be printed butted against each other. Positive numeric expressions are printed preceded by a space; negative numeric expressions are preceded by a minus sign (-).

PRINT

RANDOMIZE [RANDOMIZE] "reseeds" the computer's random number generator so that the **RND** function (see **RND** on FUNCTIONS list) does not produce the same series of numbers. Should be used before **RND** in most situations where truly random numbers are desired (as in all games).

RANDOMIZE

READ [READ var,var ...] must be used in conjunction with **DATA** (see **DATA**). Fetches item from **DATA** list, assigns to stipulated variable. Item fetched must match variable type to which item is assigned. **DATA** is fetched in sequential order from first item in line to last, from lowest numbered **DATA** line in program to highest. Computer will not use same **DATA** item twice unless **RESTORE** (see **RESTORE**) is invoked. Attempt to **READ** when **DATA** is exhausted will result in *OUT OF DATA* error and program will abort.

READ

REM [REM ...] instructs computer to ignore everything until end of line. Used as an imbedded note to the programmer. Can be read as short for **REMARK**. Must be entered with at least one space after it before comments begin or before **RETURN** is pressed.

REM

- REM ABS** **REM ABS** [REM ABS *Inum*] used in conjunction with **RENUMBER** (see Appendix A); causes line in which **REM ABS** appears to be renumbered with line number *Inum*. Subsequent lines will be renumbered *Inum* plus the base set by renumber.
- RESTORE** **RESTORE** [RESTORE] enables computer to reuse DATA previously READ. Resets DATA pointer back to first piece of DATA in first DATA statement in program.
- RETURN** **RETURN** [RETURN] causes branch to command following most recent GOSUB; pulls address of most recent GOSUB from GOSUB stack. Always final command in subroutine. Must have corresponding GOSUB or GOSUB UFLOW error will result.
- STOP** **STOP** [STOP] causes interruption in program execution, returns to COMMAND screen. Pressing [RETURN] resumes execution. Used primarily for debugging purposes.

C: Built-In Functions

The following are preprogrammed formulas built into HOB. In order to operate these formulas (called FUNCTIONS), type the function's NAME followed by (within parentheses) the number or numeric variable you want acted upon.

The following FUNCTIONS operate in either *immediate* or *deferred mode*.

For instance to find the cosine of 12 enter COS(12). You can also use numeric expressions. COS(12), COS(4*3) or COS(A) would all yield .84385396, assuming that A=12. Exceptions to these rules will be clearly noted.

Arithmetic Functions

ABS(exp) ABSOLUTE value of *exp* (i.e. value of *exp* without either + or -). **ABS (exp)**
ABS(-12) will yield 12, ABS(3*-5) will yield 15.

EXP(exp) yields the value of the constant *e* (2.718281 . . .) raised to the *exp* **EXP(exp)**
power. EXP(4) gives 54.59815, EXP(-3) yields .049787068.

INT(exp) returns the largest integer (whole number) no greater than *exp*. **INT(exp)**
INT(3.14159) will give 3, INT(-3.42) will give -4.

LOG(exp) value of the NATURAL LOGARITHM of *exp*. Value of *exp* must **LOG(exp)**
be positive. LOG(13) gives 2.5649494, LOG(3*.2) yields -.51082562.

SGN(exp) indication of algebraic SIGN of *exp*: negative gives -1, positive **SGN(exp)**
gives +1, 0 gives 0. Assuming A=37, B=-35 and C=0 then SGN(A) gives +1,
SGN(B) gives -1, SGN(C) gives 0.

SQR(exp) SQUARE ROOT of *exp*. Value of *exp* must be positive or "FUNC- **SQR(exp)**
TION ERROR" message will appear. SQR(9) gives 3.

RND returns a PSEUDO RANDOM NUMBER between 0 and .9999 . . . **RND**
NOTE: this number is not really random, but is part of a repeating series.
The **RANDOMIZE** command must be used in those situations where true
randomness is needed.

exp1^exp2 exp1 is RAISED TO THE POWER exp2. 5^3 yields 125, 2^(-5) **exp^exp2**
yields .03125.

Trigonometric Functions

ATN(exp) yields the angle whose tangent is equal to the value of *exp*. **ATN(exp)**

COS(exp) yields the COSINE of the angle whose value is equal to *exp*. **COS(exp)**
Value for *exp* must be less than 100.

SIN(exp) yields the SINE of the angle whose value is equal to *exp*. **SIN(exp)**

TAN(exp) TAN(exp) yields the TANGENT of the angle whose value is equal to exp.
Values are considered to be RADIANS unless the immediate command DEGREES is issued.

Special APPLE Functions

BTN(n) BTN(n) where *n* refers to the particular game paddle (either 0 or 1); returns a 0 if the button is not being pushed and a 1 if the button is being pushed *at the time the function is being calculated by the computer*. Using numbers other than 0 or 1 as *exp* will always yield 0.

PDL(n) PDL(n) where *n* refers to the particular game paddle (either 0 or 1); returns an integer between 0 and 255 reflecting the current position of the knob on the game paddle. Using numbers other than 0 or 1 as *exp* will always yield 0.

Defined Functions

FNvar FNvar where *var* is any single alphabetic; returns whatever FNvar has previously been defined as yielding based on the current value of *var*. If FNA had been defined as $A*5$ and the current value of A is 3, then FNA will yield 15.

FNvar(var) FNvar (var) where *var* is any single alphabetic; same as above, except that the *var* in parentheses can be any numeric variable. If the original function had been defined as $X*4$ (i.e. DEF FNX(X)= $X*4$) and currently A=5, B=7 and J=16, then FNX(A) yields 20, FNX(B) yields 28 and FNX(J) yields 64.

NOTE: Defined functions receive their definitions only from within programs. See DEF FN in Appendix A and in Chapter 9.

D: Errors and Possible Solutions

The message in **UPPER CASE** (always the first line of a section) is the error message the computer will display at the bottom of the command screen. The following line describes what the error message means. The statement(s) after the asterisk (*) are the kinds of statements that would generate the error. The comments after the colon (:) indicate possible causes for the bugs and offer suggestions for cleaning them up.

Most error messages are preceded by a listing of the line in which the error occurred.

Some Shorthand Used In These Descriptions

exp	expression. Can be any number, variable or string.	exp
int	integer. Any whole number.	int
lnum	line number. Any integer between 1 and 9999.	lnum
n	number. Any integer or decimal number.	n
num	numeric. Used to qualify meaning of other phrases.	relational
relational	one of the relational operation sets (= , < , > , <> , <= , >=).	var
var	numeric variable	var\$
var\$	string var	var\$

Static Errors

The following errors will be discovered and reported by HOB after RUN or some other SYSTEM command has been entered but before HOB actually attempts to execute your code. Thus, no variables will be initialized and no processing will be attempted until these errors have been corrected.

ARRAY var IS DIMENSIONED TWICE

named array has two separate DIM statements in the program

- * DIM var(n) or DIM var(n1,n2)
- : check if you have attempted to DIM an array after you have referred to it in some value assignment statement (thus causing automatic DIM to (10) or (10,10))
- : check if you have attempted to assign the same array title to two different arrays
- : check if you have simply attempted to DIM the same array twice
- : delete incorrect DIM statement

CANNOT BE SATISFIED

attempt to RENUMBER with illegal REM ABS Inum

- * REM ABS Inum
- : Inum too low to be accomplished
- : use higher Inum

DO WITHOUT LOOP

no LOOP statement to match corresponding DO statement

- * (deficiency)
- : check if LOOP statement was accidentally written over or deleted
- : add missing LOOP statement
- : eliminate excess DO statement

END STATEMENT MISSING

program lacks required END statement.

- * (deficiency)
- : add 9999 END

END STATEMENT SHOULD BE STOP

END was not the highest numbered line in the program; computer assumes you meant to use STOP statement instead.

- * END
- : renumber END statement to highest line number
- : check if there are more than one END statements

EXPANDS BEYOND LINE

attempt to RENUMBER will cause specified line number to contain too many characters

- * RENUMBER base, Inum
- : shorten text in program line Inum by a few characters until after successful RENUMBER; then restore deleted characters

EXPRESSION EXPANSION OVERFLOW

attempt to evaluate complex function overflows system limits

- * DEF FNvar
- * DEF FNvar1(var2)
- : simplify function so that expansion of elements within function occupies less than 250 characters

FOR WITHOUT NEXT

no NEXT statement to match corresponding FOR statement

- * (deficiency)
- : check if NEXT statement was accidentally written over or deleted
- : add missing NEXT statement
- : eliminate excess FOR statement



INDEX DOES NOT MATCH

index variable in NEXT statement does not match index variable in corresponding FOR statement.

- * NEXT var
 - : check if you mistyped index variable name in either FOR or NEXT statement
 - : check if you renamed index variable in FOR without renaming it in NEXT statement or vice-versa
 - : check if nested loops have either FOR or NEXT indexes cross-matched

LINE NUMBER OVERFLOWS

attempt to RENUMBER using specified base and increment causes line numbers to exceed maximum (9999)

- * RENUMBER base, increment
 - : lower base
 - : decrease increment



LOOP WITHOUT DO

no corresponding DO statement to match LOOP.

- * LOOP
 - : check if DO statement accidentally deleted
 - : add DO line
 - : delete excess LOOP line

MORE THAN 22 DATA STATEMENTS

attempt to have more than 22 DATA lists in same program

- * DATA var, var\$
 - : combine several DATA items into same line
 - : eliminate excess DATA list(s)

MORE THAN 4 NESTED LOOPS

FORLOOPS and/or DOLOOPS have been nested more than 4 levels deep.

- * FOR var= numexp1 to numexp2 / NEXT var
- * DO / LOOP
 - : eliminate excess levels of FORLOOPS and/or DOLOOPS
 - : rewrite excess FORLOOPS and/or DOLOOPS using COUNTERs and IF exp THEN Inum statements to accomplish same ends



NEXT WITHOUT FOR

no corresponding FOR statement to match NEXT.

- * NEXT var
 - : check if FOR statement accidentally deleted
 - : delete excess NEXT line

Inum NOT FOUND

attempt to branch to non-existent program lines.

- * GOTO Inum
- * GOSUB Inum
- * IF exp relational exp THEN Inum
- * ON var GOTO Inum1,Inum2,Inum3 . . .
- : check if you have typed correct line reference
- : add code at referenced lines

OPTION BASE MUST BE 1ST STATEMENT

attempt to declare OPTION BASE in other than lowest numbered line.

- * OPTION BASE n
- : rewrite OPTION BASE as first program line
- : eliminate OPTION BASE statement

STATEMENTS DO NOT MATCH

FOR used with LOOP or DO used with NEXT

- * FOR . . . LOOP
- * DO . . . NEXT
- : change offending DO to FOR
- : change offending LOOP to NEXT
- : change offending FOR to DO
- : change offending NEXT to LOOP

TRANSFER TO Inum IS INVALID

attempt to branch into body of FORLOOP.

- * GOTO Inum
- * GOSUB Inum
- * IF exp relational exp THEN Inum
- * ON var GOTO Inum1,Inum2,Inum3 . . .
- : check if you have typed correct line reference
- : rewrite code to avoid branch into FORLOOP

Dynamic Errors

The following errors will be discovered by HOB during program RUN time (i.e. during program execution) and will cause program execution to terminate (the program will crash) at the point where the error was encountered. Variables may have been initialized and some processing may have been accomplished. Therefore the SYMBOLS table or PROGRAM TRACKING SCREENs may contain valuable data to aid in debugging (PROGRAM TRACKING SCREENs may be accessed when program is not running by entering the appropriate letter as a control character; for instance to access the CTRACE screen enter a[CTRL][T]).

ARITHMETIC ERROR n

calculated numexp out of HOB's range

- * LET var=numexp
- : HOB will aid with further instructions at screen bottom: PRESS CR
FOR SLOW MOTION REPLAY
- : reduce var or n in exp to avoid calculating out of system's range

EXPRESSION EXPANSION OVERFLOW

attempt to evaluate complex function overflows system limits

- * DEF FNvar
- * DEF FNvar1(var2)
- : simplify function so that expansion of elements within function occupies less than 250 characters

FNvar DEFINED TWICE

attempt to redefine function with same name

- * DEF FNvar
- * DEF FNvar(var2)
- : lnum referred to in ERROR message is line where error is detected rather than the real cause of error
- : check for duplicate function name
- : check for one function defined as FNvar and another defined as FNvar(var2)
- : rename or eliminate one of the functions

FNvar NOT DEFINED

attempt to refer to undefined function

- * PRINT FNvar
- * LET var= FNvar
- : define the function
- : check for error in typing function name referenced

FNvar USED WITH WRONG # OF ARGS

attempt to refer to function (defined as having an argument) without including argument

- * PRINT FNvar [when FNvar1(var2) was called for]
- * PRINT FNvar1(var2) [when FNvar was called for]
- * LET var= FNvar
- : change FNvar to FNvar1(var2) or vice versa
- : check for error in function name referenced

FOR INDEX var IS USED TWICE

attempt to use same index variable in two active FORLOOPS

- * FOR index=numexp1 TO numexp2
- : change one index variable name

GOSUB STACK OVERFLOW

more than 16 GOSUB calls have been issued without encountering RETURN statement

- * GOSUB Inum
 - : look for accidentally deleted RETURN statement
 - : check if subroutine is calling itself
 - : RETURN statement may be numbered incorrectly
 - : add RETURN statement at appropriate line

MEMORY IS FULL, LINE NOT STORED

addition of new material to program will overflow available memory

- * (occurs when attempting to add new program line)
- * (occurs when attempting to change existing program line)
 - : DELETE or shorten REM statements to free memory space
 - : DELETE line to be changed, then re-enter it
 - : rewrite and compact sections of code using more economical constructs (for instance, use FORLOOPS instead of counter constructs)
 - : use clear abbreviations in PRINT statements

MORE THAN 4 NESTED LOOPS

FORLOOPS OR DOLOOPS nested more than 4 levels deep.

- * FOR var= numexp1 to numexp2 / NEXT var
- * DO [WHILE] [UNTIL] / LOOP
 - : eliminate excess levels of LOOPS
 - : rewrite excess LOOPS using COUNTERs and IF exp THEN Inum statements to accomplish same ends

RANGE ERROR

attempt to use array element higher than DIM'd or lower than 1 if OPTION BASE 1 invoked

- * OPTION BASE 1
- * DIM var(n) or DIM var(n1,n2)
- * LET var(n)= numexp
- * READ var(n)
- * INPUT var(n)
- * IF var(numexp)= exp THEN Inum
 - : delete OPTION BASE statement if used
 - : change DIM statement
 - : check var used to access array element
 - : check exp used to compute var to access array element

READ-DATA MISMATCH

attempt made by READ statement to fetch string from DATA list and assign it to numeric variable

- * READ var\$

- : check if DATA items have been listed in proper order
- : check if item has accidentally been added to or left out of DATA list at program entry
- : check if READ var has accidentally been added or left out at program entry
- : change READ var\$ to READ var

READ IS OUT OF DATA

all DATA lists in program have been exhausted

- * READ var or READ var\$
- : check if DATA item has inadvertently been eliminated or not entered
- : look for inadvertent READ statements
- : add DATA list
- : issue RESTORE statement

RETURN & GOSUB STACK EMPTY

RETURN statement encountered without corresponding GOSUB

- * RETURN
- : program flow has "fallen into" subroutine (add STOP statement or unconditional branch before subroutine begins)
- : program branched into subroutine via GOTO, IF-THEN or ON-GOTO statement
- : RETURN statement inadvertently entered at program entry time
- : GOSUB statement eliminated without also taking out corresponding RETURN statement

SYMBOLS TABLE OVERFLOW

more variables defined than SYMBOLS table can handle (63)

- * LET var= numexp
- * LET var\$=exp
- * READ var [or var\$]
- * INPUT var [or var\$]
- : reuse inactive variable names
- : assign infrequently used values to same variable at time of use
- : store known but infrequently used numbers in DATA statements rather than in variables and assign at time of use
- : check for "runaway" loops assigning values to array variables

TAB VALUE IS n; NOT WITHIN BOUNDS

n is 0 or greater than 255

- * PRINT TAB(n); or PRINT TAB(var);
- : set n to less than 256
- : if n set by expression check for expression computing value higher than 255

ON INDEX IS OUT OF RANGE VALUE IS n

index variable holds zero or number exceeding listed number of target
Inums

- * ON index GOTO Inum1,Inum2, . . .
- : check if there are sufficient line references after GOTO
- : check exp used to compute index

(see the DOS manual for information on the care and handling of diskettes)

The following errors occur during HOB's attempt to ROLLOUT or ROLLIN a program (DOS stands for DISK OPERATING SYSTEM). More information on these and other DOS errors can be found in the DOS manual that came with your disk drives.

DOS Errors

DISK FULL

attempt to ROLLOUT program to full diskette

- * ROLLOUT <name>
- : use another diskette

FILE NOT FOUND

attempt to ROLLIN a program that is not on the diskette

- * ROLLIN <name>
- : make sure <name> is spelled the same way it was spelled when ROLLOUT command was used
- : insert proper diskette and try again

I/O ERROR

general problems with the disk drive or diskette

- * ROLLIN <name>
- * ROLLOUT <name>
- * CATALOG
- : drive door not closed properly
- : diskette not properly inserted into drive
- : uninitialized or improperly initialized diskette in drive
- : no diskette in drive
- : diskette damaged (usually beyond repair)
- : drive malfunction

WRITE PROTECTED

attempt to ROLLOUT a program to a diskette with a write-protect tab in place

- * ROLLOUT <name>
- : remove write-protect tab from diskette and try again
- : use another diskette

E: Program Tracking Screens

This synopsis of the PROGRAM TRACKING SCREENs lists the name of each screen, the key to press to switch that screen to your monitor, and what function the screen performs. The key listed will bring that screen to your monitor while a program is running. If a program has been stopped by an ERROR, an END or STOP statement or by a press of [ESC], and if no program line has been changed, you can look at any screen by pressing and holding down the [CTRL] key before pressing the usual screen key:

screen	key	function	
COMMAND	[C]	programming, editing done from this screen; default screen when program encounters STOP or END command or when ERROR occurs.	[C]
CTRACE	[T]	displays program lines as they are executed (chronological order); shows name and value of any variable changed by that line.	[T]
DATA	[D]	shows all DATA program lines; next item to be READ displayed in inverse video; displays effects of both READ and RESTORE commands.	[D]
FORLOOP	[F]	displays for each currently active FOR-LOOP: listing of FOR and NEXT program lines, initial index value, index value limit, current index value and increment value (STEP); for each active DOLOOP DO and LOOP lines are displayed, plus values for expressions being compared in DO WHILE and DO UNTIL constructs. Up to 4 nested loops are allowed.	[F]
GOSUB	[G]	shows program lines of all active GOSUBs; lowest line displayed is deepest level.	[G]
LTRACE	[L]	displays listing (up to 19 lines at a time) of program portion currently active;	[L]

program line about to be executed is listed in inverse video (black letters on white background).

[P] PRINT

[P] displays text from PRINT statements, INPUT prompts; default screen each time program is RUN or INPUT statement is encountered.

[V] VARIABLES

[V] shows name, current value of all variables, displays line number in which value of specified variable was last changed; variables displayed 8 per "page": [+] shows next page, [-] shows previous page, [@] shows first page.

F: Summary of System Limits

NESTING LEVELS

DOLOOPS/FORLOOPS (combined)	4
GOSUBs	16
PARENTHESES	8

NUMBER OF ALLOWABLE CHARACTERS

INPUT	18
LINE OF CODE	32 exclusive of line numbers
LINE NUMBER	4
NUMBER	12 including decimals
ROLLOUT NAME	27
STRING	18

RANGES

NUMBERS	$\pm 9.99E63$ ($\pm 9.99 \times 10^{63}$)
PAGE	0-5
PDL(0) OR (1)	0-255
TAB	1-255

VARIABLES

NUMERIC NAMES
 ANY SINGLE ALPHABETIC
 A0-Z9
 ALPHABETIC(0)-ALPHABETIC(255)
 ALPHABETIC(0,0)-ALPHABETIC(255,255)
 STRING NAMES
 ANY SINGLE ALPHABETIC + "\$"

COMBINED TOTAL OF NAMED NUMERICS & STRINGS: 63
 (the DIM statement counts as 1)

NOTE: the following are all different & allowable:

A A1 A(1) A(1,1) A\$

MISCELLANEOUS

DISK DRIVES: 1 OR 2
 FILE NAMES: 1-27 CHARS. 1ST CHAR=ALPHABETIC
 CHARS 2-27: ALPHANUMERIC

NESTING LEVELS

NUMBER OF ALLOWANCE CHARACTERS

RANGES

VARIABLES

MISCELLANEOUS

G: Glossary of Terms

This list contains brief definitions for most of the words found in the TERMS section at the end of each chapter, plus other words used throughout the manual. Where there is more than one definition for a word the definitions are separated by semicolons. If you can't find a definition for a particular word here, check the index or try one of the other appendices (ERRORS, FUNCTIONS, DEFERRED STATEMENTS, etc). Words appearing in *upper case* without quotes within a definition are themselves defined within this glossary:

ACTIVE	ACTIVE defined within a program (when referring to a <i>variable</i>); currently being executed (when referring to a section of <i>code</i>).
ADDRESS	ADDRESS location in computer's memory where a particular piece of information can be found; used as a verb to mean "reference" or "speak to" or "discover the contents of" (particularly when dealing with a <i>variable</i>).
ALGORITHM	ALGORITHM step by step procedure for solving a problem.
ALPHABETIC	ALPHABETIC letter of the alphabet.
ALPHANUMERIC	ALPHANUMERIC alphabetic or number 0–9.
ARRAY	ARRAY list of variables referenced by the same <i>array title</i> . Each <i>variable</i> may be addressed independently by referring to the number of that variable's <i>element</i> .
ARRAY TITLE	ARRAY TITLE the <i>alphabetic</i> used as the name of an <i>array</i> .
BODY	BODY <i>code</i> that falls between the first and last lines of a <i>loop</i> and which may therefore be <i>executed</i> more than once.
BRANCH	BRANCH transfer <i>execution</i> from one part of a computer program to another; the command that causes such a transfer.
BREAKPOINTS	BREAKPOINTS invisible markers set by BREAK command causing <i>execution</i> to halt temporarily at specific program lines.
BUG	BUG a human-introduced <i>code</i> error that causes unexpected, usually unwanted, occasionally humorous results in computer programs and insanity in computer programmers.
CODE	CODE all or any portion of a computer program.
COMMAND	COMMAND an instruction to the computer to take some action.
CONTROLLED LISTING	CONTROLLED LISTING method of <i>listing</i> a program using the [SPACE] bar so that the speed at which the computer displays and erases program lines is not too fast for the computer operator to read.

COUNTER a <i>variable</i> used to keep count of something; especially, the form $X=X+1$ in which a <i>variable</i> is incremented by 1 each time a specific event occurs.	COUNTER
CR short for <i>carriage return</i> ; also written $\langle CR \rangle$.	CR
CRASH to make a program come to an abrupt, unanticipated, unwelcomed halt due to some error.	CRASH
CTRACE abbreviation for CHRONOLOGICAL TRACE, one of HOB's program tracking screens.	CTRACE
CURSOR flashing white square on display screen indicating where next character will be displayed; sometimes a <i>prompt</i> for the operator to type something.	CURSOR
DATA list of items used by a READ command and assigned to <i>variables</i> within a program; also the values of all <i>variables</i> .	DATA
DEBUG the process of tracking down and correcting errors.	DEBUG
DEFERRED one of the two modes of <i>execution</i> (the other being <i>immediate</i>); specifically refers to commands that are part of a computer program and which will not be carried out by the computer until another, special command is issued (specifically RUN).	DEFERRED
DEFINE assign a value to.	DEFINE
DELAY LOOP type of loop designed to "kill time" while some other action is occurring (typically a screen display) and to put off the execution of the next program section.	DELAY LOOP
DELETE erase from the computer's memory one or several program lines; erase from the <i>diskette</i> an unwanted program.	DELETE
DISK alternate form of <i>diskette</i> ; occasionally can refer to <i>disk drive</i> (will be clear in context).	DISK
DISK DRIVE device which stores and retrieves computerized information.	DISK DRIVE
DISKETTE medium upon which computerized information is stored by a <i>disk drive</i> .	DISKETTE
DOLLAR (\$) prettily colored piece of paper rumored to have once been worth something; also refers to the sign for STRING .	DOLLAR (\$)
DOS abbreviation for Disk Operating System, the commands and special code that allow programs to be ROLLED OUT or ROLLED IN .	DOS
EDIT to modify a program line or immediate command line.	EDIT
ELEMENT a specific variable in an <i>array</i> , usually referenced by an <i>integer</i> or	ELEMENT

another *variable* between parentheses after the *array title* [as in A(12) or B(N)].

ERROR **ERROR** state of the program in which *execution* is halted by other than a STOP or END command or [ESC].

ERROR STATEMENT **ERROR MESSAGE** text displayed on computer monitor notifying operator of error condition and giving pertinent information about error's possible location in the program.

EXECUTION **EXECUTION** the carrying out of a set of commands.

EXPRESSION **EXPRESSION** any group of numbers and/or *variables* coupled with operators and/or function, meant to be taken as a single value.

HANG **HANG** a verb describing an error condition in which the program is "spinning its wheels" with no *prompt*, no change in the value of *variables*, no commands operating, and without program control returning to the command screen.

HARD COPY **HARD COPY** a program listing or text written on paper by either a printer or a person.

HOB **HOB** an abbreviation for HANDS ON BASIC.

HUMANIZE **HUMANIZE** to make fit for use by humans; to write programs taking into consideration the needs, likes, dislikes, age, educational level and general condition of the person or persons likely to use it.

IMMEDIATE **IMMEDIATE** one of two modes of execution (the other being deferred); specifically refers to commands carried out by the computer as soon as the [RETURN] key is pressed.

IMMEDIATE COMMAND **IMMEDIATE COMMAND** command carried out in *immediate mode*.

INITIALIZE **INITIALIZE** assign an original value to (as in INITIALIZE A VARIABLE); prepare to accept data (as in INITIALIZE A DISK).

INTEGER **INTEGER** a whole number; number with fractional or decimal part removed.

INTERACTIVE **INTERACTIVE** type of program that seeks information from the computer operator and typically contains one or a number of input commands.

INVERSE VIDEO **INVERSE VIDEO** black characters appearing on a white or green background (usually on a computer screen white or green characters appear on a black background).

LINE NUMBER **LINE NUMBER** an *integer* between 1 and 9999 appearing at the beginning of every HOB program line.

LISTING a sequentially numbered series of coded instructions which, taken as a whole, constitute a BASIC computer program.	LISTING
LOAD to assign values to the <i>elements</i> of an <i>array</i> ; to fetch a program from a diskette and place in computer memory.	LOAD
LOOP a programming construct consisting of several lines of <i>code</i> whose action is repeated any number of times.	LOOP
LTRACE abbreviation for LIST TRACE, one of HOB's program tracking screens.	LTRACE
NUMERIC dealing with numbers (as opposed to alphabetics or special characters).	NUMERIC
O a flashing symbol at the top of HOB's various program tracking screens alerting the operator that text is being displayed on the print screen.	O
OPERATOR person using computer; arithmetic symbol describing how various values in an expression are to be combined.	OPERATOR
OVERFLOW a condition whereby some upper limit within the HOB system has been exceeded.	OVERFLOW
PADDLES pair of rheostatic control devices that plug into the computer and allow the operator to send the computer a range of numbers (between 0 and 255) depending on the setting of the rheostats.	PADDLES
PRECEDENCE order in which arithmetic operations will be carried out.	PRECEDENCE
PROGRAM set of coded instructions that make a computer operate.	PROGRAM
PROGRAM TRACKING SCREEN any one of five special displays in HOB used as aids in teaching programming or to help in the debugging process.	PROGRAM TRACKING SCREEN
PROMPT LINE instructions on a computer monitor saying what information is to be entered from the computer's keyboard or how such information is to be entered.	PROMPT LINES
REFERENCE LINES program lines to which the computer will branch (found within GOTO, GOSUB and IF-THEN commands).	REFERENCE LINES
RELATIONAL OPERATORS the symbols < (less than), = (holds the contents of), and > (greater than) so called because they describe the relationship between two variables or between variables and a number.	RELATIONAL OPERATORS
RETURN the [RETURN] key on the computer keyboard which, when pressed, signals the computer that the information entered is to be acted upon in some way; also the final program line in any subroutine causing the computer to branch back to the program line following the GOSUB command initiating the subroutine action.	RETURN

SINGLE STEP	SINGLE STEP to proceed through a computer program in chronological order one line at a time (accomplished in HOB by pressing the [SPACE] bar).
STACK	STACK list maintained by HOB of program line numbers from which currently active subroutines were called.
STRING	STRING any group of characters enclosed in double quotes (""); any group of characters not consisting entirely of numerics.
SUBSCRIPTED VARIABLE	SUBSCRIPTED VARIABLE another name for <i>element</i> .
TARGET	TARGET <i>referenced line</i> in a GOTO, GOSUB or IF-THEN command; used in <i>error messages</i> to indicate a non-existent <i>reference line</i> .
TEXT	TEXT any characters displayed on the computer monitor or meant to be displayed via a PRINT command.
UNDERFLOW	UNDERFLOW condition whereby some lower limit within the HOB system is exceeded; usually refers to an error in which a <i>return command</i> has been encountered for which there is no corresponding GOSUB.
VARIABLE	VARIABLE a symbol or combination of symbols used to represent any one of a number of values and whose represented value can change within a program through the use of various commands and arithmetic operations.
WRITE-PROTECT TAB	WRITE-PROTECT TAB an adhesive (usually metallic) label placed over a small notch on a diskette's upper right edge; prevents programs from being ROLLED OUT to, or deleted from, that particular diskette.

H: Editing Keys and Commands

The following keys and commands can all be used to modify instructions or program lines entered into HOB. The terms EDIT and DEL are both command words to be followed by specified line number(s); all other terms refer to specific keys on the APPLE keyboard:

EDIT <i>Inum</i>	prepare to EDIT existing line	EDIT <i>Inum</i>
[←]	move cursor over character to left	[→]
[→]	move cursor over character to right	[←]
[CTRL][I]	insert blank space at cursor; move all characters to right of cursor to right (I for INSERT)	[CTRL][I]
[CTRL][O]	delete character under cursor; move all characters to right of cursor one place to left (O for OMIT)	[CTRL][O]
[CTRL][X]	cancel current line and erase from screen; does not change program line in memory (X for X-OUT)	[CTRL][X]
DEL <i>Inum</i>	remove line <i>Inum</i> from program	DEL <i>Inum</i>
DEL <i>Inum1</i>,<i>Inum2</i>	remove all lines between <i>Inum1</i> and <i>Inum2</i>	DEL <i>Inum1</i>, <i>Inum2</i>
<i>Inum</i>	same as DEL <i>Inum</i>	<i>Inum</i>
[REPT]	repeats last character pressed; holding down any key while pressing [REPT] will repeat that key's action indefinitely.	[REPT]

I: Precedence

The following chart describes HOB's order of evaluation in solving all problems and formulae. Note that expressions within parentheses are solved first; in the case of nested sets of parentheses, expressions within the innermost sets are evaluated first. Expressions on equal levels are evaluated from left to right. Items higher up on the chart have evaluation precedence (i.e. are evaluated first):

()	PARENTHESES from innermost nested to outermost	()
+n or -n	SIGNED ARITHMETIC	+n or -n
^	EXPONENT	^
exp*exp or exp/exp	MULTIPLY or DIVIDE	exp*exp or exp/exp
exp+exp or exp-exp	ADD or SUBTRACT	exp+exp or exp-exp

J: Solutions

The following are proposed solutions to the problems presented at the end of chapters 1–9. They represent one or two POSSIBLE solutions for each problem; they are not the ONLY solutions. If your solutions follow the rules of syntax, adequately define a term, or describe a situation appropriately then your solution is just as valid as any of the ones listed here.

Chapter 1 Chapter 1

1. Your COMMAND LINE (what you should type on the screen after the prompt and before you press the RETURN key) should look like this:

$(6+2)*3^3$

The order of evaluation will be $6+2$ (8), 3^3 (27) and $8*27$. The $6+2$ gets evaluated first because of the parentheses. The final answer will be 216. If you forgot the parentheses you'll probably get 60 ($27*2+6$) for an answer.

2. There are at least two possible answers here:

$(5-3)/.5$ or $(5-3)/(1/2)$

Either one will yield 4, the correct answer.

3. The order of evaluation here will be $15-4$ (11), 2^3 (8), $11*8$ (88), $5-88$ (-83), and $-83+83$ (0). Once again, the parentheses around the $15-4$ forced HOB to evaluate that expression first.

FINETRACE <CR> turns on FINETRACE; NOFINETRACE <CR> turns it off.

Chapter 2 Chapter 2

1. LET A=6 <CR>
LET B=.2 <CR>
LET Q=A-(B*30) <CR>
LET S=A/Q <CR>
2. A <CR>
B <CR>
Q <CR>
S <CR>
3. SYMBOLS <CR>
4. LET B=B+3

If B hadn't been initialized yet an error message would have appeared.

5. N is a numeric variable and the phrase "SOMETHING'S WRONG" is a string. Strings can ONLY be assigned to string variables (any alphabetic followed by a \$). A correct form would have been
`LET N$="SOMETHING'S WRONG"`
6. `NEW <CR>`

Chapter 3

Chapter 3

Budding Genius Solution: Line 20 is executed only 14 times. The value for X is incremented in line 12 and is checked by line 18. If the value for X is 15 in line 18 the program branches out of the loop; line 20 gets executed one less time than the rest of the lines in the loop because the branch occurs BEFORE line 20 is executed.

1. `10 LET Z=0`
`20 LET Z=Z+1`
`30 PRINT "HERE"`
`40 IF Z=10 THEN 60`
`50 GOTO 20`
`60 PRINT "THERE"`
`70 END`
2. (REUSE 10 & 20)
`30 PRINT Z`
`40 IF Z=10 THEN 70`
 (REUSE 50 AND 70)
3. Line 40 sends the computer back to line 10 which resets the value of Q to 0. The value of Q alternates between 0 and 1; the program is in an infinite loop. Change line 40 to
`40 GOTO 20`
4. "If variable Q holds the value 5 then branch to line 50".
5. Line 30 is a conditional branch; the computer will go to line 50 ONLY if the value held by variable Q is 5. Line 40 is an unconditional branch; whenever the computer reads this line it will immediately branch to line 10.
6. RUN will execute a program; LIST will show in sequential line order all the commands in the program.
7. To store the program, first give the program a name. Then issue the command "ROLLOUT <name>". Finally, follow the directions on the screen. To recall the program issue the command "ROLLIN <name>".

Chapter 4 Chapter 4

1.

```
10 FOR Z=6 TO 12
20 PRINT Z;
30 NEXT Z
40 END
```
2.

```
10 FOR Z=12 TO 6 STEP -1
```
3. To make things clearer, think of S as representing SUM and N as representing NUMBER:

```
10 LET S=0
20 FOR N=1 TO 50
30 LET S=S+N
40 NEXT N
50 PRINT "THE SUM IS";S
60 END
```

Another way to do the same thing is to use a COUNTER:

```
10 LET S=0
20 LET N=0
30 LET N=N+1
40 LET S=S+N
50 IF N=50 THEN 70
60 GOTO 30
70 PRINT "THE SUM IS";S
80 END
```

4. The code is the same for both versions:

```
PRINT N;S
```

The line number would be 35 for the first version or 45 in the second version.

Chapter 5 Chapter 5

1. This program has a MISMATCH in line 20. Attempting to READ E will crash the program; the numeric variable E is the fifth variable in the READ list, but the fifth item in the DATA list is the string variable MARTHA. E must be changed to E\$.
2. Set PACE to 75 before RUNNING the program, press [←] four times immediately after issuing the RUN command, or add a nested DELAY LOOP (as it turns out, with a *highlimit* of 20) inside the N1 FORLOOP. The first two methods cause program execution to slow down to about 1 line/second; the third method works by "stalling" normal program execution for some period. All these methods work for APPROXIMATE time periods; don't set the timer on your life-support system's WARNING bell by them.

3. Use READ . . . DATA and a set of QUOTES:

```
10 DATA "THIS IS IT,NO?"
20 READ A$
30 END
```

4. 10 DATA REPETITIOUS

```
20 READ A$
30 RESTORE
40 READ B$
50 RESTORE
60 READ C$
70 RESTORE
80 READ D$
90 RESTORE
100 READ E$
110 END
```

or

```
10 DATA REPETITIOUS
20 READ A$
30 LET B$=A$
40 LET C$=A$
50 LET D$=A$
60 LET E$=A$
70 END
```

After the program RUNs, type SYMBOLS and press [RETURN]. Single-step through the program by pressing the [SPACE] bar immediately after issuing the RUN command (remember that in most cases a command isn't really "issued" until you press [RETURN]); press [SPACE] bar to execute one line at a time. Press [T] to get the CTRACE screen.

5. 10 DATA A,B,C,D,E,F,G,H,I,J,K,L,M,N
20 DATA O,P,Q,R,S,T,U,V,W,X,Y,Z
30 FOR N1=1 TO 26
40 READ L\$
50 PRINT L\$,N1
60 NEXT N1
70 END

OR

```
5 LET N1=0
10 DATA A,B,C,D,E,F,G,H,I,J,K,L,M,N
20 DATA O,P,Q,R,S,T,U,V,W,X,Y,Z
30 DO UNTIL N1=26
```

```

35 LET N1=N1+1
40 READ L$
50 PRINT L$;" +3";N1
60 LOOP
70 END

```

Chapter 6 Chapter 6

1. 10 PRINT "HI! WHAT'S YOUR NAME?"
 20 INPUT N\$
 30 PRINT "OK," ;N\$;
 40 PRINT "YOU'RE LOOKIN' GOOD!"
 50 END

2. 10 PRINT "PLEASE SET PDL(0): ";
 20 FOR Z=1 TO 100
 30 NEXT Z
 40 LET A=PDL(0)
 45 PRINT A
 50 PRINT "TIME TO RESET PDL(0): ";
 60 FOR Z=1 TO 100
 70 NEXT Z
 80 LET B=PDL(0)
 85 PRINT B
 90 PRINT "FINAL RESET:";
 100 FOR Z=1 TO 100
 110 NEXT Z
 120 LET C=PDL(0)
 125 PRINT C
 130 PRINT
 140 PRINT "THE SETTINGS WERE";
 142 PRINT A;" ";B;" ";C

Another way to do the delays between paddle "samplings" is to use INPUTS. Instead of lines 20 and 30, for instance, you might use

```

20 PRINT "TYPE 'OK' AND PRESS RETURN";
25 PRINT " WHEN READY"
30 INPUT A$

```

Line 30 is a "dummy" INPUT. We don't really care what A\$ ends up having assigned to it; we just use it as a way of making the computer wait until the operator is ready to go on.

3. Line 20: A should be A\$.
 Line 30: PDL(5) doesn't exist!
 Line 30: no space between "WELL," and A\$ (poor formatting)
 Line 50: END statement must be last line in program

4. A TARGET line and a REFERENCED line are essentially the same; HOB uses the term TARGET in error messages. Both terms refer to lines to which the computer will BRANCH from commands like GOTO and IF-THEN. A PROMPT line is a sentence or phrase displayed on the screen before an INPUT statement telling the operator what he or she is supposed to type.
5.


```

10 PRINT "THIS PROGRAM WILL MULT";
20 PRINT "IPLY THE SET-"
30 PRINT "TING ON PDL(0) BY THE ";
40 PRINT "ONE ON PDL(1).";
50 PRINT "SET EITHER PADDLE TO";
60 PRINT " 0 TO END."
62 FOR W=1 TO 200
64 NEXT W
65 PRINT
70 LET A=PDL(0)
80 LET B=PDL(1)
90 LET C=A*B
100 PRINT A;" TIMES";B;" EQUALS";C
110 IF C=0 THEN 130
120 GOTO 65
130 PRINT
140 PRINT
150 PRINT "SEE YOU AROUND!"
160 FOR W=1 TO 100
170 NEXT W
180 END
      
```

Chapter 7

Here's the addition to the ARRAY J(A,B) problem:

```

80 FOR B=1 TO 3
90 LET J(0,B)=0
100 FOR A=1 TO 4
110 LET J(0,B)=J(0,B)+J(A,B)
120 NEXT A
130 NEXT B
      
```

1. A numeric variable can have only one value while an ARRAY can have a whole series of numeric variables (called ELEMENTS) with the same name (called the ARRAY TITLE).
2.


```

10 FOR P=1 TO 5
20 READ J(P)
30 NEXT P
      
```

Chapter 7


```
40 DATA 53,19,60.5,4,17
999 END
```

3. The contents of J(3), which is 60.5, multiplied by the contents of J(4), which 4, will produce 242. There are at least two ways of storing this number in J(6):

```
50 LET J(6)=J(3)*J(4)
```

or

```
50 LET J(6)=242
```

4. ARRAY Q has a total of 60 possible elements. Since the DIM statement appears in line 1 the OPTION BASE must be 0 (if an OPTION BASE isn't declared in the first line of a program, its default value is zero; the lowest line number possible is already being used). Allowing for the 0 element in both dimensions, multiply (5+1) times (9+1) or 6 times 10 yielding 60.
5. To eliminate the possibility of any 0 elements the lowest numbered line in the program would have to contain the command:

```
OPTION BASE 0
```

In the listed example the line number for the DIM statement would have to be higher than it is; line number 1 is the lowest line number possible, and the OPTION BASE command must appear as the first command in the program.

Chapter 8

Chapter 8

1. HOB would give a RANGE error. The input variable A6 contains the value 9; the same variable is used as an index for the ON-GOTO command in line 30. But line 30 has only 4 reference line numbers (100, 297, 116 and 999). Since 9 is more than 4, the index value 9 is out of range.
2. The string "THIS IS A TEST" would be printed once, and the string "HERE'S THE TESTED PART" would be printed 16 times. Then the program would crash with GOSUB STACK OVERFLOW. The problem is caused by line 110; it sets up a loop consisting of lines 20, 100 and 110. The loop would be infinite if the stack didn't overflow, causing the program to abort.
3. RUNning this program would cause a RETURN & GOSUB STACK EMPTY error at line 120, meaning that a RETURN has been encountered without a corresponding GOSUB. A trace of the program would show the following line execution sequence: 10—20—100—110—120—30—100—110—120—CRASH. When the program RETURNS the first time, execution continues to line 30 and then "falls" into the subroutine at line 100. There

needs to be a line of "protection" code to block off the subroutine from accidental entry:

40 GOTO 999

4. These are called RELATIONAL because they make comparisons between two variables to see how they RELATE to one another (i.e. is A greater than B, less than B or equal to B).
5. The following is our solution to problem #5; it's a bit fancier than it needs to be. If your program fulfills the requirements and if it doesn't crash, then your program is fine! Programming is an art — and there's room for much creativity in solving a particular problem. Spaces between the sections in the following listing are for convenience in reading and understanding; they CANNOT be "installed" in your own listings. The REM statements are commands for the programmer and are not executed by the computer; see Chapter 9:

5 REM AGE GUESS PROGRAM

6 REM INITIALIZE VARIABLES

7 REM FIRST, PROGRAMMER'S AGE

10 LET A=37

20 REM NOW THE GUESS COUNTER

30 LET C=0

35 REM HEADING AND DIRECTIONS

40 PRINT TAB(6);"GUESS MY AGE ";

45 PRINT " WITHIN 5 GUESSES!"

50 PRINT "TYPE IN A NUMBER & ";

60 PRINT "PRESS THE RETURN KEY"

62 REM INPUT LOOP STARTS HERE

65 PRINT

70 INPUT I

80 LET C=C+1

90 IF I=A THEN 500

95 IF C=5 THEN 700

100 IF I<A THEN 130

110 GOSUB 300

120 GOTO 65

130 GOSUB 200

140 GOTO 65

150 REM INPUT LOOP STOPS HERE

200 REM GUESS TOO LOW

210 PRINT

220 PRINT "NO — I'M OLDER!"

230 RETURN

```
300 REM GUESS TOO HIGH
310 PRINT
320 PRINT "I'M YOUNGER THAN THAT!"
330 RETURN

500 REM CORRECT GUESS
510 PRINT
520 PRINT "YOU GOT IT! IT TOOK YOU";
530 PRINT C; " GUESSES!"

550 REM ROUTINES TO CHEER "WINNERS"
560 ON C GOTO 600,620,640,660,680
600 REM 1 GUESS

610 PRINT "THAT'S GREAT GUESSING!"
615 GOTO 995
620 REM 2 GUESSES
630 PRINT "PRETTY GOOD GUESSING!"
635 GOTO 995
640 REM 3 GUESSES
650 PRINT "THAT'S NOT BAD!"
655 GOTO 995
660 REM 4 GUESSES
670 PRINT "AN OK SCORE!"
675 GOTO 995
680 REM 5 GUESSES
690 PRINT "RIGHT UNDER THE WIRE!"
695 GOTO 995

700 REM A LOSER!
710 PRINT
720 PRINT "SORRY — YOUR GUESSES ";
730 PRINT "ARE ALL USED UP!"
740 PRINT "HAVE ANOTHER GO AT IT!"

910 REM DELAY LOOP
995 FOR Z=1 TO 250
996 NEXT Z
9999 END
```

Chapter 9 Chapter 9

Writing this program was sort of a “final exam”; you really had to use what you’ve learned about BASIC to put it together. Like other “solutions”, this final one is only a POSSIBLE way to do it. Yours might be far different — and far better!

Spaces between the sections in the following listing are for convenience in reading and understanding; they CANNOT be "installed" in your own listings:

```
1 REM DICE GAME PROGRAM
2 REM SET UP VARIABLES
3 REM FIRST FOR THE POINT
4 LET P=0
5 REM NEXT FOR THE SUM OF DICE
6 LET T=0
7 REM AND THE DICE THEMSELVES
8 LET R1=0
9 LET R2=0
10 REM SET UP DICE FUNCTION
11 DEF FND=INT(6*RND)+1
12 REM GIVE INSTRUCTIONS
14 PRINT TAB(15);"DICE GAME"
20 RANDOMIZE
22 PRINT
24 PRINT "PRESS THE BUTTON ON PDL";
26 PRINT "(1) TO ROLL"
28 PRINT
40 PRINT

45 REM GET PLAYER'S NAME
47 REM TO "PERSONALIZE" THE GAME
50 PRINT "WHAT'S YOUR NAME, ACE?"
60 PRINT "(ENTER 'END' TO QUIT)"
70 INPUT N$
72 REM HERE'S AN ESCAPE
75 IF N$ "END" THEN 9999
80 PRINT "OK, ";N$;" , YOUR DICE!"
90 PRINT "SHOOT FOR YOUR POINT!"

92 REM FIRST THROW
95 REM "DICE THROW" SUBROUTINE
100 GOSUB 1000
105 REM "FIRST THROW" WINNERS
110 IF T=7 THEN 5000
120 IF T=11 THEN 5000
125 REM "FIRST THROW" LOSERS
130 IF T=2 THEN 6000
140 IF T=3 THEN 6000
150 IF T=12 THEN 6000
153 REM IF GAME GOES ON,
```

```
154 REM GET THE "POINT"
155 LET P=T
160 PRINT
170 PRINT "OK, ";N$;" , ";
180 PRINT "ROLL THEM BONES!"

185 REM MAJOR LOOP STARTS HERE
190 PRINT " YOUR POINT'S ";P
195 REM ROLL THEM BONES!
200 GOSUB 1000
205 REM IS IT A WIN?
210 IF T=P THEN 5000
215 REM IS IT A LOSS?
220 IF T=7 THEN 6000
225 REM NEITHER — ROLL AGAIN!
230 GOTO 190
235 REM THAT ENDED MAJOR LOOP
237 REM AND "PROTECTED" THESE
239 REM FOLLOWING ROUTINES
```

```
997 REM ROUTINE SECTION
1000 REM ROLL THE DICE
1005 PRINT
1010 PRINT "YOUR ROLL:"
1015 REM BUTTON PRESS LOOP
1020 IF BTN(1)=1 THEN 1040
1030 GOTO 1020
1035 REM HERE'S THE FIRST DIE . . .
1040 LET R1=FND
1042 REM . . . AND THE SECOND.
1045 LET R2=FND
1047 REM WHAT DO THEY TOTAL?
1050 LET T=R1+R2
1060 PRINT "YOU ROLL";R1;" AND ";R2
1070 PRINT TAB(15);"THAT'S ";T
1075 PRINT
1080 RETURN
```

```
5000 REM WINNER'S MESSAGE
5001 REM LET THE BELLS PEEL
5002 FOR Z=1 TO 50
5004 BEEP
5006 NEXT Z
5010 PRINT N$;" "S A WINNER!"
5020 PRINT "LOOK OUT, LAS VEGAS!"
```

```
5030 GOTO 9995
6000 REM LOSER'S MESSAGE
6010 PRINT "TOO BAD, ";N$;" . A LOSS."
6020 PRINT "GIVE IT ANOTHER SHOT!"

9000 REM DELAY LOOP
9995 FOR Z=1 TO 250
9996 NEXT Z
9999 END
```

K: Single Key Commands

These single key commands will have their stated effect DURING program execution. Commands marked with an asterisk (*) will also work during a program interruption if they are entered as control characters. Also see APPENDIX H, EDITING commands

[C]*	call COMMAND SCREEN	[C]*
[D]*	call DATA SCREEN	[D]*
[ESC]	interrupt execution	[ESC]
[F]*	call DOLOOP SCREEN	[F]*
[G]*	call GOSUB SCREEN	[G]*
[L]*	call LIST TRACE SCREEN	[L]*
[N]	click speaker every other command	[N]
[P]*	call PRINT SCREEN	[P]*
[Q]*	call query line (program status)	[Q]*
[RESET]*	crash system	[RESET]*
[RETURN]	set pace to full (5)	[RETURN]
[S]*	print "photo" of current display	[S]*
[SPACE] bar	set pace to step (0)	[SPACE] bar
[T]*	call CHRONOLOGICAL TRACE SCREEN	[T]*
[V]*	call VARIABLES SCREEN	[V]*
[X]	set pace to 5; switch to PRINT SCREEN	[X]
[;]	call loopstep	[;]
[,]	call gosubstep	[,]
[+]*	page forward in VARIABLES SCREEN	[+]*
[-]*	page backwards in VARIABLES SCREEN	[-]*
[@]*	recall first page in VARIABLES SCREEN	[@]*
[←]	slow program pace (down to 0)	[←]
[→]	speed program pace (up to 5)	[→]

INDEX

Most error messages are not indexed here; see Appendix D. System limits are covered in Appendix F; editing keys are in Appendix H.

Special Symbols

127
\$ 25, 27, *see also* string,
 see also variable
* 13
+ 13, Appendix K, 26
- 13, Appendix K, 26
/ 13
; 45, Appendix K
 as LOOPSTEP
 command key, 128
<CR> *see* CR
= 23, 27, 36
> meaning "ready to
 accept command",
 13
? 64,
 suppressing, 65
@ 26, Appendix K
^ 17, Appendix C
 defined, 17
 found on keyboard,
 17

A

ABS (exp) 20, Appendix C
absolute branch *see* branch,
 unconditional
active 27, Appendix G
addition 13
address Appendix G
aesthetics 66, 67
algorithm 111, Appendix C
alphabetic 27, Appendix G
APPLE FUNCTIONS
 Appendix C
applications program 109
arithmetic error 19
ARITHMETIC FUNCTIONS
 Appendix C
array 74, 114, 115–116,
 Appendix G, *see also* DIM

chart 75
name vrs. value 75
rules 82
title Appendix G
ATN (exp) 20, Appendix C
audio feedback 131

base 68
BEEP 100, Appendix B
BEEPHI Appendix B
BEEPLO Appendix B
bel *see* BEEP
body 56–57, Appendix G
booting system 9
branch 34, Appendix G
 conditional 36, 84
 IF/THEN 34
 GOSUB/RETURN 89
 ON/GOTO 86–87
 unconditional 36, 90
BREAK 125–126, Appendix A
BREAKFIND 126, Appendix A
BREAKLIST 127, Appendix A
BREAKPOINT 126–127,
 Appendix G
BREAKPOINT marker 127
BTN (n) 96, 99–100, Appendix
 C, *see also* paddles
bug 33, 119, Appendix G
bytes 131

C Appendix K
CARRIAGE RETURN *see*
 RETURN
CATALOG 32, Appendix A,
 see also DELETE,
 see also ROLLOUT
chained 16

CHRONOLOGICAL TRACE
 SCREEN 56, Appendix E
 printed on paper 129
 showing values of
 variables 122–123
code 31, Appendix G
 combining lines of 45
comma Appendix K
 as GOSUBSTEP command
 key 128
 for TAB FIELDS 113
command Appendix G
COMMAND SCREEN
 Appendix E
 returning to 33
conditional branching 84
CONTROL 35
control loops 43
controlled listing Appendix G
controlling execution
 speed 56–57
COS (exp) Appendix C
 limit of 20
counter 24, 34, 46, 55,
 Appendix G
 reset to 0 37
CR Appendix G
 defined 15
 to *see* value of variable 23
crash 60, Appendix G
CTRACE *see*

B

C

CHRONOLOGICAL TRACE
 SCREEN
CTRL *see* CONTROL
cursor 15, Appendix G
 defined 14

D Appendix K
DATA *see also* READ/DATA,
 Appendix B, Appendix G

D

list 55, 58–59, 61–62
total number allowed 62
DATA SCREEN 59, Appendix E
debug Appendix G
debugging tools 119
DEF FN 96, 102, Appendix B
deferred Appendix G
defined 31
with syntax errors 32
DEFERRED COMMANDS
LIST Appendix B
define Appendix G
definition see variable
DEGREES 21, Appendix A
DEL 45–46, 55–56
delay loop 48, 100, Appendix G
DELETE 40, Appendix A,
Appendix G
program line 45
DIM 75–76, *see also* array
Appendix B
automatically set to 10
76–77
disk Appendix G
CATALOG 39
DELETE 40
initialize 38
list of programs stored 39
LOCK 40
preventing accidental
deletions 40
remove programs from
diskette 40
retrieving programs 39–40
reusing 40
ROLLIN-ROLLOUT 38
ROLLOUT 38
storing programs with 2
drives 39
two drives 39
UNLOCK 40–41
disk drive Appendix G
using int-3
diskette Appendix G

DO 51, Appendix B
DO UNTIL 51, Appendix B
DO WHILE 52, Appendix B
dollar 25, Appendix G
DOLOOP 50–51
DOLOOP SCREEN see
FORLOOP SCREEN
DOS Appendix G
DYNAMIC ERROR
CHECKING 33
EDIT 44, Appendix A,
Appendix G
chart of commands 46
insert & delete characters 45
restrictions 46
sneaky 89
editing keys and commands
(chart) Appendix H
element 74, *see also* array
Appendix G
each as separate variable 74
END 33, Appendix B
error 119, Appendix G
message Appendix G
OUT OF RANGE 88
protection 88
READ IS OUT OF DATA 60
READ-DATA MISMATCH 59
STACK FULL 93
uflow 92
ESC 33, Appendix K
during INPUTS 65
to discontinue listing 68
use described 37
execution Appendix G
EXIT 131, Appendix A
exp 18
EXP (exp) Appendix C
expression Appendix G

E

F Appendix K
FIND Appendix A
FINETRACE 25, 97–98,
Appendix A
automatic 19
defined 16
F displayed on PROGRAM
STATUS LINE 97–98
in an exercise 22
shutting off 16
FNvar Appendix C
FOR-TO-STEP Appendix B
FORLOOP 43
index value passing
highlimit 56
SCREEN 48
used to load array 75
FORLOOP SCREEN Appendix E
abbreviations explained 49
DOLOOPS on 52
format 113
FULL 58
function
arithmetic 20
arithmetic with 24–25
built-in 19–20
defined by programmer
102–103
how errors are handled 21
how to operate 19–20
optional study of 22
trigonometric 21
FUNCTIONS LIST Appendix C
G Appendix K
game paddles see paddles
GOSUB Appendix B
GOSUB SCREEN 90,
Appendix E
GOSUB/RETURN 89
GOSUBSTEP 128
GOTO 36, Appendix B

F

G

hang Appendix G
hard copy 129, Appendix G
HOB Appendix G
 getting up and running int-3
 how to get the most out of
 int-2
hidden programs 90–91,
 131–132
HOME 113, Appendix B
humanize Appendix G

I

IF/THEN 34, 36, Appendix B
 branch *see also* conditional
immediate Appendix G
 defined 31
IMMEDIATE COMMANDS
 LIST Appendix A
increment 68
increment 35–36
index 43
 bad match error 49
 ON/GOTO variable 88
 stack (GOSUB) 92–93
infinite loop 33–34, 38, *see*
 infinite loop
initialization 110
initialize 23, 35–36, *see also*
 THE DOS MANUAL
 diskettes 38–39, Appendix C
INPUT 64, 110, Appendix B
 and “humanized
 programming” 65
 program interruption
 during 65
 rules 64
INT (exp) Appendix C
INT (n) 96, 97
integer 97, Appendix G
interactive 65, Appendix G
inverse video 85, Appendix G

H

L Appendix K
left arrow 14–15, 32,
 Appendix K
 found on keyboard 14
 to control program speed 57
LET 23, 27, Appendix A,
 Appendix B
line number Appendix G
line numbers 31
 automatic 32
 unintentional 32
 indented 32
 limits 32
LIST 34, 43, Appendix A
 keys that control 68
LIST TRACE SCREEN 85,
 Appendix E
listing Appendix G
LLIST 128–129, Appendix A
load Appendix G
loading HOB into computer
 int-3
LOCK 40, Appendix A
LOG (exp) Appendix C
LOOP Appendix B,
 Appendix G
 body of 48
 control 43
 delay 48, 100
 DOLOOP 50–51
 FORLOOP 43
 infinite 33–34
 major 36, 55
 nested 49
LOOPSTEP 128
LTRACE Appendix G

memory cells 131
mixed 19

N 17, 131, K

L

nesting 17, 49
 subroutines 90–91
NEW 26–27, Appendix A
NEXT 43, Appendix B
NOFINETRACE 16–17,
 Appendix A
NOISE 131
numeric Appendix G

O

O (flashing) 57, Appendix G
ON var GOTO Appendix B
ON/GOTO 86–87
operations *see* operator
operator 13, 16, 17,
 Appendix G
 for division 13
 taught in most schools 13
 used in programming 14
OPTION BASE 80, *see also*
 array, Appendix B
 placement in program 80–81
output 110
overflow Appendix G

P

P Appendix K
PACE 58, Appendix A
 reset to 5 via RETURN 58
 reset to 5 via X 58
 using SPACE BAR to set to
 STEP (Ø) 56
paddles 68, 99–100,
 Appendix G
 using to assign value to
 variables 71
parentheses 16–17, 24–25
PART 58
PDL (n) 70, *see also* paddles,
 Appendix C
 using to assign value to
 variables 71
planning programs 109
playing computer 84–85

powers of numbers 18
PR#n 130, Appendix A
precedence 16, 17–18,
 Appendix G
 chart of 17, Appendix I
 exponents level of 18
 in setting execution
 speed 57–58
 parentheses and 16–17
PRINT 33–34, Appendix B
 on paper 130
 series of variables on same
 line 45
 to create a blank screen
 line 66–67
 with **TAB (n)** 66–67
PRINT SCREEN 35, Appendix E
 ON notice 57
 recalling 43
PRINTER COMMANDS 128
PRINTER IS IN SLOT n 130,
 Appendix A
 process 110
 program 31, Appendix G
 applications 109
 controlling speed of
 execution 57
 crash 60
 internal notes for 98–99
 phases 110
 planning 109
 size 131
 specifications 109
 program lines *see also* line
 numbers
 adding 34
 printed on paper 128–129
 recycling 56, 66–67
 setting **BREAKPOINTS** at 127
PROGRAM STATUS LINE 92
PROGRAM TRACKING
SCREEN 48, 79, 85, 90,
 Appendix G
 how to access 124

inaccessible after changes 50
 numbers at top of 49
ROLLED OUT with
 program Appendix A
 synopses of all Appendix E
 prompt line 65, 110,
 Appendix G
 prompt lines 66
 pseudorandom numbers 96
PUN 58, Appendix A

Q Appendix K
QUERY 49, 91

RADIANS 21, Appendix A
 random number 96
RANDOMIZE 96, 99,
 Appendix B
READ/DATA 55, Appendix B
 and string variables 59
 rules for 61
 referenced line 68, Appendix G
 relational operators 88,
 Appendix G
 chart of 36
 displayed on **CTRACE** 123
REM 98, Appendix B
REM ABS Inum 101,
 Appendix B
RENUMBER 68, Appendix A
 default values 68
REPEAT key 89
REPT 89
 required equipment int-1
RESET Appendix K
 resident language 131–132
RESTORE 60, *see also* **READ/**
DATA, Appendix B
 summarized 61
RETURN 1, *see also* **CR**,

Q

R

Appendix B, Appendix G
 cause full-speed execution 57
 continuing execution
 interrupted by **BREAK** 126
 continuing program stopped by
ESC 36–37
 location of key 14–15
 must press for most
 commands 32
 not needed in **CONTROL**
 sequences 35
 to continue listing at full
 speed 68
 with **FINETRACE** 16
 with **INPUT** 64
RIGHT ARROW 46,
 Appendix K
 to control program speed 57
RND 96, Appendix C
 assigning generated value to
 variable 97
ROLLIN 40, Appendix A
ROLLIN-ROLLOUT diskette 38
ROLLOUT 38, Appendix A,
see also **DELETE**,
see also **CATALOG**
RUN 32–33, Appendix A

S

S 130, Appendix K, Appendix A
SAVE *see* **ROLLOUT**
 scientific notation 18
SCREEN
CHRONOLOGICAL
TRACE 56, Appendix E
COMMAND 26–27,
 Appendix E
DATA 59, Appendix E
FORLOOP 48, Appendix E
GOSUB 90, Appendix E
LIST TRACE 85, Appendix E
PRINT 35, Appendix E
VARIABLES 120, Appendix E

screen snapshot 131
seed number 99
SGN (exp) Appendix C
SIN (exp) Appendix C
single step 56, Appendix G
SIZE 131, Appendix A
SPACE BAR Appendix K
 for automatic line numbers 32
 for suspending execution 56
 single step with 56
 to control program listing 68
speed of execution 56–57
 chart 58
 keeping same 58
 X for full speed 58
SQR (exp) Appendix C
stack 93, Appendix G
STEP 44–45, 59, *see also*
 FORLOOP
STOP Appendix B
string 25, 27, *see also* variable,
 Appendix G
 defined 25–26
 length 27
subroutine defined 92–93
subscript 74, *see also* element
subscripted variable 74, *see*
 also array, *see also* variable
subtraction 13
SYMBOLS 26, *see also*
 VARIABLES SCREEN,
 Appendix A

syntax error 19
 defined 16
 in deferred mode 32
system requirements int-1

T Appendix K
tab field 113
TAB (n) 66–67
TAN (exp) Appendix C
target Appendix G
text 45, Appendix G
TR#n 129, Appendix A
TRIGONOMETRIC
 FUNCTIONS Appendix C
TRY 32

underflow Appendix G
UNLOCK 40, Appendix A

V Appendix K
var *see* variable
variable Appendix G, *see also*
 SYMBOLS, *see also*
 VARIABLES SCREEN
array 74
 assigning values to 75
 defined 23
changing values of 27

clearing all 27
compared to Algebraic
 24–25
defined 23
locating via FIND 124
numeric 23
 arithmetic with 24
 assigning value to
 23, 68, 71, 97
 defined 23
 rules for 27
SCREEN 120
string 25
 defined 23
 rules for 27
 setting value to 25
strings and READ/DATA 59
table of 26
to see value of 23, 120
value kept 23–24
watching changes through
 CTRACE 56
VARIABLES SCREEN 120,
 Appendix E, *see also*
 SYMBOLS

write-protect tab Appendix G

X Appendix K

W
X